20

A TRIDENT SCHOLAR PROJECT REPORT

NO. 201

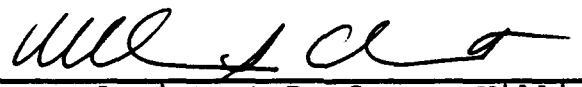
"A Stereoscopic Vision System with Applications to Automated Docking and Tracking"

UNITED STATES NAVAL ACADEMY
ANNAPOLIS, MARYLAND

U.S.N.A. - Trident Scholar project report; no. 201 (1993)

"A Stereoscopic Vision System with Applications to Automated Docking and Tracking"

by
Midshipman Michael M. Hsu, Class of 1993
U.S. Naval Academy
Annapolis, Maryland



Adviser: Assistant Professor William I. Clement
Department of Weapons and Systems Engineering

Accepted for Trident Scholar Committee

Chair

May 17, 1993

Date

REPORT DOCUMENTATION PAGE

Form Approved
OMB no. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 17, 1993	

4. TITLE AND SUBTITLE

A stereoscopic vision system with applications to automated docking and tracking

5. FUNDING NUMBERS

6. AUTHOR(S)

Michael Ming Hua Hsu

7. PERFORMING ORGANIZATIONS NAME(S) AND ADDRESS(ES)

U.S. Naval Academy, Annapolis, MD

8. PERFORMING ORGANIZATION REPORT NUMBER

U.S.N.A. - Trident scholar project report ; no. 201

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

Accepted by the U.S. Trident Scholar Committee

12a. DISTRIBUTION/AVAILABILITY STATEMENT

This document has been approved for public release; its distribution is UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

A current project of great import to the National Aeronautics and Space Administration is the development of an automated docking and tracking system to facilitate autonomous operations in space. Such a system would have to be able to determine the relative positions, velocities, and orientations of a multitude of bodies.

A stereoscopic vision system was developed to implement an automated docking solution. This system used artificial neural networks to identify beacons or fixed points on the objects to be tracked. Specifically, this research sought to solve the problems inherent in the planned 1995 docking of the U.S. Space Shuttle to the former Soviet Space Station Mir.

In addition to the basics of an optical ranging and object-recognition system, a simple user interface for operation monitoring was designed. Specifically, the coordinates of the space station, shuttle waypoints, and smooth trajectory position, velocity, and acceleration information were calculated and displayed. A mock docking was also simulated, with the vision system providing the range and orientation data. By placing the vision system at known coordinates and checking its computed trajectory, the accuracy of the algorithms and given hardware were checked.

14. SUBJECT TERMS

Space docking and tracking systems, stereoscopic vision systems, object-recognition systems, automated systems

15. NUMBER OF PAGES

77

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED

Abstract

A current project of great import to the National Aeronautics and Space Administration is the development of an automated docking and tracking system to facilitate autonomous operations in space. Such a system would have to be able to determine the relative positions, velocities, and orientations of a multitude of bodies.

A stereoscopic vision system was developed to implement an automated docking solution. This system used artificial neural networks to identify beacons or fixed points on the objects to be tracked. Specifically, this research sought to solve the problems inherent in the planned 1995 docking of the U.S. Space Shuttle to the former Soviet Space Station Mir.

In addition to the basics of an optical ranging and object-recognition system, a simple user interface for operation monitoring was designed. Specifically, the coordinates of the space station, shuttle waypoints, and smooth trajectory position, velocity, and acceleration information were calculated and displayed. A mock docking was also simulated, with the vision system providing the range and orientation data. By placing the vision system at known coordinates and checking its computed trajectory, the accuracy of the algorithms and given hardware were checked.

Table of Contents:

Note: The majority of the ideas and algorithms presented in this paper were developed either in collaboration with or solely by Assistant Professor William I. Clement.

Motivation and Purpose

Currently, the policy of maintaining humans in space is called into question by the cost of transporting the essentials for human survival into space. Other than for reasons of medical research and publicity, humans in space are neither vital nor economical at this time. The use of robotics to accomplish some human missions could drastically cut the costs of future operations in space exploration and development of space for humanity. One prerequisite to practical robot operations in space is a flexible sensor system with which more complete automation can be accomplished. Systems using the spectrum visible to the human eye are the sensors most conducive to operational systems designed to mimic human functions.

The use of robotics is definitely not limited to space. Aerial or even undersea robotic systems would require the same type of visual sensory data to be truly effective. Furthermore, automation enhanced by machine vision has already revolutionized some aspects of our lives. The automated car assembly line operated by robots is one example.

However, before truly effective multi-purpose robots can be made, they must be given the essential tools through which they can have awareness of their surrounding

environment. Of the many sensor technologies available, vision will be the most important because of its flexibility. Robots must be able to 'see' the objects around them, determine if and how they are moving, and identify them as well. Once a robot can open its 'eyes', analyze its surroundings, and identify the objects in its environment, it will be capable of performing innumerable tasks.

The ultimate goal of this Trident project was the design of a vision system capable of locating, tracking, and recognizing objects in its environment. These goals may be pursued to widely varying degrees of refinement and success. The technology explored by this project will have vital uses in future exploration and development in all realms - atmospheric, terrestrial, space, and aquatic. The automated docking and tracking system described here is a substantial first step to such a system.

Approach

In the docking situation simulated, a space shuttle must rendezvous and dock with a space station, using a stereoscopic vision system connected to its flight computer as a sensor. The space shuttle has already been maneuvered to the general vicinity of the station, and to the general area of the side of the station with which it is to dock.

Figure 1 shows a nearly completed docking between the space shuttle and a futuristic space station.

The system was designed to isolate the basic problems encountered in a stereoscopic vision system. This project was a simplification of the general case by limiting the area in which the system had to search for its target, and limiting the target to a specific beacon of known and unique characteristics. These restrictions were imposed in part by the hardware used, as seen in the field of view of the camera, and in part by lack of sufficient progress made thus far in some vision technologies, such as edge detection. Later work was directed towards the optimization of this system, and development of algorithms to fine-tune the tracking process.

The majority of the work on this project was done in computer code. The final product was not hardware, but a set of algorithms in the form of a program, which was adaptable to more advanced hardware than that of the current system setup. All algorithms, including the interaction with the frame-grabbing and frame-processing boards and the software-based artificial neural network, were written in the 'C' programming language.

The automatic range-finding algorithms were relatively straightforward. The goal of the investigator was to automate this process to the greatest extent. Further development of the project was initially planned to

incorporate the development and use of specific geometries of visual beacons which could enhance the ranging process and help automate the finding of corresponding points on each image. This approach was later deemed unnecessary, as there already exist on space stations many distinctive objects which could be isolated by vision systems.

Instead, the use of neural networks and pattern recognition was incorporated to identify the beacons, and even the objects themselves with speed and accuracy. The goal of this project was the production of a system which, when given a target, would automatically lock onto it and determine its range, bearing, orientation, and identity, and compute waypoints and a trajectory to it.

Why the Optical Approach? Advantages and Problems

The optical approach to this problem was superior to the radar approach. First, at close proximity, radar interference with objects and with other similar systems caused significant problems. In this case, the passive nature of the optical solution was also superior. An optical system was far more effective at tracking a multitude of objects at one time. It was limited mainly by its field of view and computer processing speed. Second, the detail accommodated by optical systems allowed much more precise mapping of objects in order to determine object

orientation. The detail provided by visual data could also be exploited to develop a pattern recognition approach to object identification and/or categorization.

While the development of a docking system was the primary goal of this project, the system has a variety of other capabilities. One is the capability to track a multitude of smaller objects, such as the tools and components of a space structure under construction. Further refinement of this vision system to enhance tracking is facilitated by the current approach. By using a pattern recognition system based on a simple computer-based neural network, objects on the space station (for orientation purposes) and independent objects in proximity of the system can be identified. This combination of capabilities allows an automated system to have awareness of its environment; a prerequisite to fully automated operations in space.

There were also several problems with vision systems which were either briefly considered in this project or relegated to ongoing vision research. These issues included interference (in the form of glare), image improvement, and edge detection. The first two obstacles were found to be partially solved by the use of the artificial neural network approach. The interpolating nature of solutions achieved by artificial neural networks compensates for some degree of error and noise interference in object descriptors. However, accurate edge detection, a subject of extensive

research, will be vital to the success of this project.

Obviously, use of this system is precluded in regions where sunlight was either unavailable or eclipsed by the space station itself. Spotlights could partially solve this problem. Examination of current space operations shows similar limitations. The astronauts must have light to see and the sun must not be in a position which blinds them. Therefore, this system could at least be used in the majority of docking situations already encountered.

The Mir Space Station

The Mir Space Station was launched on February 20, 1986. The name, meaning "Peace", signified the Soviets' opposition to the Strategic Defense Initiative. [1] Mir's commonality with its predecessors, the Salyut stations, was determined primarily because of the dimensions of its launch vehicle, which were the same. However, Mir featured a significant improvement: six docking ports, rather than the two of the Salyut. While Salyuts had docking ports at each end of their cylindrical structures, Mir had four more ports spaced around the node of its forward transfer compartment. [2] Its 20 metric ton initial form was designed to be expanded by means of these ports to a 120 ton, six-module configuration, some of which is shown in Figure 2.

To date, three additional modules have been added. The

Kvant-1 ("Quantum") astrophysics module was docked to the stern port, with some difficulty, on April 12, 1987. This 11-ton module incorporated advanced equipment from the Netherlands, Great Britain, the European Space Agency, and West Germany, in addition to Soviet equipment. Kvant-2 (20 tons) was manually docked to one of the four forward radial ports in December, 1989. Twenty-ton Kristall, known as the T-module, was added to the opposite radial port in 1990. Additionally, Kristall, a materials processing and biotechnology module, was to function as a docking port for the Soviet Buran space shuttle and other future international manned spacecraft. This module is equipped with the Androgynous-Peripheral Docking System (APDS-89), which was based on the system used in the U.S. - U.S.S.R. Apollo-Soyuz Test Project of 1975. One of the U.S. Space Shuttles will dock with this module in 1995. [3]

In addition to the four current modules, Mir is occasionally visited by Prognoz M ("Progress") and Soyuz TM ("Union") vehicles, which respectively provide the Salyut and Mir space stations with resupply and crew transfer capabilities. Each weighs seven tons. When the U.S. docking occurs with Mir, the entire station complex will weigh between 71 and 85 metric tons. [1,2,3] This mass, coupled with the U.S. Space Shuttle's variable mass, (between 73 and 102 metric tons) poses a difficult docking problem to solve because of the large momenta. Both

vehicles are moveable under their own power. Only one, however, the shuttle, will maneuver independently. Previous U.S. and Russian docking operations have relied on manual operation of the vehicles, involving visual cues or radio signals. Examples include the Igla and newer Kurs rendezvous and docking systems. [3]

The U.S. Space Shuttle

The U.S. Space Shuttles were designed with the capability to dock with other objects in space. Within their 4.6 by 18.3 meter cargo bays [4], the shuttles can carry L-shaped docking ports which lie flat at launch and rotate to a vertical protruding position upon commencement of docking operations. [5] Access to and from the space shuttle is available at the aft of the mid deck, through a 2.1 meter high airlock, which has an inside diameter of 1.6 meters. The aft flight deck station of the shuttle serves as the command center for rendezvous maneuvering, and has hand controls and attitude indicators for that purpose. In addition, the shuttle has a pair of windows facing out of its top, and a pair of windows facing aft into its cargo bay from which crew members can supervise and control docking and robot manipulator operations. [4]

To provide thrust and attitude control in orbit, space shuttles are equipped with two Orbital Maneuvering System

(OMS) Engines to the left and right of the upper main engine, 38 primary Reaction Control System (RCS) jets (14 in the nose, and 12 each in the OMS pods), and 6 vernier RCS engines (2 in the nose, 2 each in the OMS pods). The OMS engines provide a thrust of 6,000 pounds for major trajectory changes (they can be swivelled 8 degrees in any direction), the primaries 870 pounds, and the verniers 25 pounds for fine control. The shuttle computer can fire these engines to make large orbital changes for rendezvous, or maintain precision pointing with an accuracy of less than half a degree. [4,5]

The Shuttle Camera System

At all four corners of the 15 by 60 feet cargo bay are surveillance cameras with a nominal purpose of monitoring the robot manipulator's operations with the payload. These four cameras are monitored from the aft flight deck at the payload operator's station. The two cameras placed nearest this station are shown in Figure 3. In addition, selection processes are currently underway to choose surveillance cameras for use on board Space Station Freedom for both monitoring and docking purposes. Cameras similar to these will likely be mounted on either side of the docking assembly within the shuttle bay as well to further aid rendezvous operations. This assortment of cameras,

totalling six will include the existing four in the shuttle bay. They provide an excellent sensor array for a stereoscopic docking system, with minimal adjustment of the available hardware. To maximize existing hardware to accomplish stereo vision and object recognition, this project simulated the use of camera pairs already on board the shuttle.

Experimental System Components

The system used in this investigation consisted of:

(1) Two Panasonic Mini-Cameras with variable apertures and assorted lenses. The lenses used have focal lengths of 7.5 mm.

(2) A Unisys 80386 Computer with 80387 Math Co-processor and the "C" Programming Language.

(3) The Data Translation DT2867 frame grabber and frame processing board.

(4) The Data Translation DT2878 high performance image and signal processing board.

(5) The "NeuralWorks Professional II/PLUS" neural network software, which provides a software-based artificial neural network.

An illustration of the setup is found in Figure 4.

System Setup

The baseline setup for basic ranging functions placed the cameras a variable distance apart, depending on the probable distance of the object to be ranged. For initial calculations, the cameras were set up on a stand which kept them at a four inch separation. An experiment to determine the optimum distance between the cameras for variable ranges was found to be unnecessary. In general, wider camera separation and greater resolution of the cameras will provide greater stereo ranging accuracy. However, a combination of wide camera separation and limited field of view can cause problems for stereo ranging and identification of objects very close to the camera array. One solution to the variable range/variable resolution problem adds a third camera and allows the user to switch between camera pairs. By thus selecting the pair, optimal results may be attained under varying conditions.

The actual hardware custom-built by NASA to accomplish the docking solution would be reduced in size to a box even smaller than the 80386 computer actually used to conduct experimentation. This box would be connected to the cameras within the shuttle payload bay in much the same configuration as the current system setup. Some means for the Shuttle's on-board computer to activate the docking system would be necessary to initiate acquisition and tracking of the space station. Once the hard-wired

algorithms for stereo vision, object recognition, location and orientation determination, and waypoint and trajectory computation were executed, some means of transferring the calculated information to the Shuttle's on-board computers would also be necessary.

Basic Ranging Problem

In order to allow the computer to range an object, a camera constant, in units of image pixels per inch of surface along the imaging array at the back of the camera, had to be determined. Only then could algorithms be written to calculate the distance based on the correspondence of images from the two cameras. The number of pixels the image displaced from one camera to another, combined with the pixels per inch constant, enabled the computer to determine the effective length of the apparent image separation in length units rather than pixel units.

The pixels per inch of the cameras was determined by a basic experiment which used only one of the cameras and assumed that both were reasonably similar. The focal length was already known to be 7.5 millimeters. A target of known width and distance from the camera was taken, and a one-camera ranging equation was used to determine how long of a stretch of the Charge-Coupled Display (CCD) the image occupied. This equation operated on the premise of similar

triangles. Tne distance to target and length of target was one set of variables, and the focal length and actual length of subtended CCD was the other (Figure 5).

$$\frac{\textit{Target Length}}{\textit{Distance to Target}} = \frac{\textit{Length of Subtended CCD}}{\textit{Focal Length}}$$

By simply using a program to count the number of pixels subtended by the object, the pixels per inch were determined by:

$$\textit{Pixels per inch} = \frac{\textit{Number of Pixels Subtended by Target}}{\textit{Length of Subtended CCD}}$$

A rough estimate of the camera variable was found by repeated experiments using a four inch (10.16 cm) target at a distance of 32 inches (81.28 cm). The target subtended 98.25 pixels in the camera image. The value of the constant was found to be approximately 2,655 pixels/inch.

Stereo ranging utilizes the mathematical relationship among focal length, distance to target, camera separation, and image disparity (parallax) of the two cameras. The general equation relating these four variables is derived using similar triangles as shown in Figures 6 and 7. [6]

Stereo Ranging and Supporting Variable Equations:

$$\frac{\textit{Distance to Target}}{\textit{Camera Separation}} = \frac{\textit{Distance to Target} + \textit{Focal Length}}{\textit{Length of Subtended CCD}}$$

$$\textit{Length of Subtended CCD} = \textit{Camera Separation} + \textit{Image Parallax}$$

$$\textit{Image Parallax} = \textit{Number of Pixels Offset} \cdot \textit{Inches per pixel}$$

In order to accomplish stereo ranging with the system used, it was necessary to determine the effective parallax between two images of a single object in both cameras. First, the object was isolated using image binarization. Binarization transforms the image from one containing many shades of gray to a black and white image based on a certain threshold (gray-level value). This gray-level is determined by observing the gradient in gray-level values between the object and its background. A simple method for determining the threshold is to use the mean of the gray-levels found for the object and background. For convenience in initial experimentation, black and white objects and backgrounds were used. The large gray-level gradients they provided simplified the binarization process.

In calculating image parallax between the two cameras, object centroids were chosen as corresponding points. The

centroids were determined using the method of chain coding. Chain coding is an algorithm which traverses the perimeter of a binarized object, thus allowing for the calculation of its mass moments and perimeter. The object's centroid may be determined from its mass moments. Once the centroid is found, the number of pixels of offset between the two images is calculated as the horizontal displacement of one image relative to the other. This corresponds to the centroid column value differences in both cameras. The distance to the designated object was then found using the stereo ranging equation. For this experiment, only horizontal correspondence was used. Vertical or even diagonal correspondence could be easily used if it was proven necessary to accommodate the dual camera configuration. With the cameras in a side-by-side configuration, only horizontal correspondence was necessary.

Determination of Bearing to Objects

Once stereoscopic ranging had been accomplished, the next step involved determination of bearings to target objects. The azimuth and elevation to the target were necessary to compute the location of the beacon in 3-D space. The easiest method of computing the offset of the beacons from the cameras was observation of the displacement of the centroid of the beacon from the centerline of the

camera. Unlike ranging, which required two cameras, bearing computation could be done with just one.

For determination of the centerline pixels, the experimental program simply took the average of the minimum and maximum row numbers. Computation of the centerline columns was conducted in the same way. This method assumed identical pixel dimensions along each row, and orientation of the camera such that the light detection elements were aligned perfectly normal to the centerline. The centerline passed through the middle of the detection array. With actual cameras on the shuttle, checks will have to be done to ensure this is the case.

Instead of computing the azimuth and elevation, the program computed the distances along the x- and y-axes, the range computed using stereoscopic vision, the focal length, and the number of pixels offset from the centerline. Figures 8 and 9 illustrate the relationships used to determine bearing. The distances were computed according to the following formula, which was derived from the stereo ranging equation:

$$x\ offset = \frac{\#\ Pixels\ Offset \cdot Inches/Pixel \cdot Distance\ to\ Target}{Focal\ Length}$$

In this fashion, the x, y, and z coordinates of each beacon are determined, as illustrated in Figure (10). The next

problem lies in computation of the Roll-Pitch-Yaw orientation of the space station relative to the space shuttle.

Determination of the Orientation of the Space Station

An initial orientation experimentation was performed using three light targets marked in white on a black background. This polarity shift between object and background made it necessary to incorporate the analysis of light objects on dark backgrounds in the chain-coding routine. The three targets were positioned to form a Cartesian coordinate frame, with beacon 1 at the origin and beacons 0 and 2 along the x- and y-axes. The z-axis of this frame coincided with the approach direction for docking. This arrangement was chosen to allow complete determination of the space station's orientation, as seen in Figure (11). These three vectors, normalized to unit length, were arranged into a 3-by-3 rotational transformation matrix. This matrix held all of the orientation information necessary to compute the roll, pitch, and yaw of the three-beacon structure relative to the docking platform.

Determination of Waypoints and Trajectory

Once the orientation and coordinates of the space

station are known relative to the shuttle, the waypoints which the shuttle should pass through or near on its path to the docking port may be calculated (See Figure 12). The first waypoint is the shuttle's current location. The second is a point straight ahead of the shuttle at a distance which would allow it twice the time necessary to rid itself of any current translational or rotational velocity (i.e., come to a complete stop). The third point is similarly located in front of the space station docking port, but at a distance to allow three times the minimum deceleration time (to force a slow approach). The fourth and fifth points are at the docking port itself. This duplication provides for a complete stop in the trajectory calculations.

Richard P. Paul [7] provides some simple equations to accomplish the smoothing of trajectories given the waypoints through which a robotic end effector, or a shuttle in this case, must pass. A smooth trajectory provides more efficient use of energy since complete accelerations and decelerations at each point are unnecessary. The shuttle need only pass near most of the waypoints.

Trajectory Algorithm:

$$acceleration = \frac{1}{2\tau^2} \cdot (\Delta C \cdot \frac{\tau}{T} - \Delta B)$$

$$velocity = \frac{1}{\tau} \cdot (\Delta C \cdot \frac{\tau}{T} - \Delta B) \cdot h + \frac{\Delta B}{\tau}$$

$$position = [(\Delta C \cdot \frac{\tau}{T} - \Delta B) \cdot h + 2\Delta B] \cdot h + Initial\ position$$

These equations are used with each set of three consecutive waypoints. The position, velocity, and acceleration refer to each of the x, y, and z coordinates and pitch, roll, and yaw orientations of the shuttle. ΔB is the difference between the parameters at the first and second waypoints, while ΔC is the difference between the second and third. Tau (τ) is the transition time before and after the central waypoint state over which acceleration occurs. T is the total time over which each transition and region of linear motion occurs. H ($0 < h < 1$) is a counter to allow computation of the position, velocity, and acceleration at any point in the trajectory (0 is the beginning, and 1 is the end of the trajectory leg). The entire trajectory over five waypoints may be calculated using two loops of the described equations. Leg one would include waypoints one, two, and three, and Leg two would include waypoints three, four, and five. The duplication of points four and five provide for a complete stop at the docking port. [7]

Automation of Calibration and Ranging Algorithms:

Determination of the pixels per inch scale factor of one camera was accomplished through the use of an interactive program between user and vision system. Variations on this approach, using a computer "mouse" as input, could be implemented on the deployable unit. This problem requires the determination of the number of pixels subtended by an object of known width at a known range. The edges of the target can be determined by finding the two sharpest gradients in the selected region. The region is chosen such that these correspond to the left and right edges of the object. For the two cameras mounted behind the crew deck of the shuttle, an example of a suitable object is the large vertical stabilizer prominently mounted in the cameras' field of view. After counting the pixels, the algorithm would have the user enter the width of the target and the distance to it, or it could use known values in the case of the fixed vertical stabilizer. The pixels per inch constant will generally remain unchanged for any given camera, assuming negligible expansion of camera imaging elements from post-launch heating of the shuttle orbiter. The incorporation of these automatic calibration routines mostly assists in the 'portability' of this software docking system to include other camera systems.

Automatic calibration of the alignment between the two

cameras (only accounting for horizontal plane cant) could also be accomplished. This algorithm would allow the user to determine the limiting columns between which the calibrating target is located, and determine the edges of the target in each camera by finding the two sharpest gradients in the designated regions.

Correlation would be used to find corresponding edges in both images. By using a loop and a 3,5, or 7-pixel segment from one camera, preferably centered around an edge, it could be slid along the same row of pixels from the other camera. Based on the user's prediction of the corresponding location, a reduced area could be searched. The point of maximum correlation would be the location of the correspondence point between the two images. This information, along with the user-supplied range to target, is enough to determine the yaw angle of one camera relative to the other.

Compensation for Camera Irregularities

Because of the "fishbowl" effect caused by the distortion from the cameras' lenses, only the central portion of the screen was used for computations. The whole camera area could be utilized once the extent of the "fishbowl" effect on measurements near the edge of the screen had been determined. It would require the development of a screen masking system or formula which

would take into account the increased distortion at the limits of the camera's view.

Visual Beacons

Current NASA approaches to its spacecraft docking and tracking problems utilize visual beacons. Visual beacons must be contrasted against the background of the docking object in order to be easily located by eye. Visual beacons must also provide necessary ranging and orientation information to the approaching spacecraft, in order to allow the operator to perceive the necessary trajectory to achieve docking.

This research incorporated the development and use of visual beacons to simplify the tracking problem. If the system were required to identify the space station, and home in on it, problems would be encountered once the approaching spacecraft closed within a certain distance of the target. The camera system would not be able to visualize the entire object even if it were mounted at the extreme aft areas of the spacecraft.

A more logical approach involves using the tracking system to identify the space station at longer distances, and to identify the visual beacon, and home in on it at shorter distances. A possible criteria for the initiation of the switch between space station tracking and visual

beacon tracking would be the percentage of the screen the space station filled. Once the station filled half the screen, the system would start searching the image for the visual beacon.

To facilitate the search for and use of the visual beacon, the design of the beacon itself is important. There should be great contrast between the beacon and its background (the space station). Beacons could be of unique and simple geometric design, to allow the vision system to identify it with simpler pattern recognition algorithms. The unique design and arrangement of the beacons must also convey all necessary range and orientation information in detail to the tracking system.

The current visual beacon used by NASA is three-dimensional. Simply described, it is a pole projecting from a circular background. The tip of the pole is one color, while the background is another. If the approaching spacecraft is not coming in on a straight trajectory, the tip of the pole will appear offset within the circular background, indicating the direction and degree of correction necessary to restore the proper approach path. Range is indicated by the apparent size of the circular background, based on a scale built into the video systems of the approaching spacecraft.

The stereoscopic nature of the proposed system eliminates the need for prior knowledge of the dimensions of

the beacon. However, thus far, knowledge of the relative orientation of the two objects still requires some type of three-dimensional beacon structure. A series of two-dimensicnal beacons would be preferable, as they could be painted or placed more conveniently at any docking location.

Part of the experimentation in this research involved the development and evaluation of scaled versions of two-dimensional visual beacons.

Holographic Beacons

A fascinating possibility is that of a holographic visual beacon. Given the advances in holography and LED lasers, it is feasible to have a flat plate of holographic film laid over a bed of Light-Emitting Diode (LED) lasers. As spacecraft approached the docking location, the visual beacon could be turned on. The holographic beacon could possibly contain the image of a ten-foot pole projecting into or out of the spacecraft. A beacon of this nature would provide very accurate information on the orientation of the spacecraft while taking up minimal space.

Discussion of Pattern Recognition Approach

Among the attributes of a desirable visual beacon would be distinctive characteristics which would make it easily

distinguishable to a machine vision system. These include different orders of moments about the centroid of the image of the beacon. If the size of the beacon were known, then the observed range and size of the object could be incorporated into the pattern recognition criterion. After using histograms, binarization, and edge-detection techniques to isolate high-contrast objects within the field of vision, the system would analyze each to determine whether its moment characteristics matched those of the desired visual beacon.

If the average albedo (image lightness or darkness) of the space station were known, which it would probably be, then the system could normalize the image operated upon based on that value. Assuming also that the albedo of the visual beacon were known relative to the space station, one possible approach to quickly identify the beacon would be to choose its color whereby reflectivity and location in the spectrum were unique.

This approach is hampered by a fundamental difficulty: the variable lighting of the sun. On the day side of the earth, reflection would require dampening filters. On the night side, searchlights might be necessary to provide enough information to the vision systems. An alternative is the use of infrared vision systems, active or passive, to

find the beacon. However, this approach will not be covered by the scope of this project.

Neural Networks for Object Recognition

Because of the uncertainties involved in the correlation of unknown object parameters to those of the beacon, an artificial neural network could greatly speed the identification process. An artificial neural network is, according to Robert Hecht-Nielsen, "...a computing system made up of a number of simple, highly interconnected processing elements, which processes information by its dynamic state response to external inputs." [8] An illustration of an individual artificial neuron model is found in Figure (13). Using the various momenta of the objects as input parameters, a trained artificial neural network would output the degree to which the unknown object matched the characteristics of the known beacon.

This project investigated the effectiveness of varying numbers of parameters in a neural network in identifying a visual beacon from among a pool of unknown objects. Other than moments, neural networks could also search for other patterns, possibly allowing the use of distinctive, intricate designs as the framework of the beacon. Software based neural networks are available for use on personal computers, and can be incorporated into the C routines

already being used to interact with the frame-grabbing and frame-processing boards.

Identification of Beacons using Neural Networks

Differentiation among the three beacons (0, 1, and 2) was necessary for proper determination of the orientation of the station. The preferred method of distinguishing objects was an artificial neural network approach. Artificial neural networks using four of invariant moments of Hu [9] and compactness (perimeter^2/area) as inputs proved to be an effective way of classifying the digits zero, one, two, and three from inclinations of up to 60 degrees from horizontal. A simple demonstration of this capability was performed on a separate camera system, and the resulting neural network subroutine was incorporated into the trajectory-computation program. Training of the neural network was accomplished using simple duplicates of the actual beacons. These targets were photographed repeatedly using a camera system. Their invariant moments and compactness were calculated, and the value of the digits broken down into binary representation by two bits of 0 and 1. The resulting data was saved to a disk file from which the training program retrieved the information. Therefore, a five input, two output neural network was used. One hidden layer neuron proved to be sufficient in this case to distinguish between

these simple digits. Figure (14) illustrates a schematic of the applied artificial neural network.

This approach assumed that beacons similar to the ones used in experimentation would have to be placed onto the space station, either before launch or during an extra-vehicular activity afterwards. Given that the Mir space station is already in place, and that extra-vehicular activities to place such beacons on the station would be costly and probably inaccurate in their results, an alternative method of fixing the station's coordinates and orientation was found to be necessary. The three-beacon approach was still feasible. However, instead of subsequently placed targets specially designed for docking operations, existing structures on the station were to be used.

The same neural network could have been trained to recognize individual objects integral to the construction of the space station. Examples include the solar arrays, an antenna, or even the name of the vehicle, if it were emblazoned prominently on the side of the spacecraft. Given that the computer had prior information fixing the locations of these identified objects in a given three-dimensional pattern on the station, the orientation of the station could be calculated in the same method used to determine orientation of the previously discussed three-beacon structure.

Neural Network Trainers and Expert Systems

Further flexibility of this system could be accomplished by combining the artificial neural network training scheme with an artificial intelligence "expert system". Combined with a simple database, this format would have a picture inventory of prominent objects to be found on any number of space vehicles. With input from what aspect and which vehicle the shuttle was to rendezvous, the autonomous docking system could choose objects for use as beacons, formulate their two-dimensional images as determined from their three-dimensional computer models, and compute their predicted characteristics.

Horizons, Vision, and Conclusion

This research covers many of the fundamentals of machine vision. Stereo vision and the location and tracking of an object in three-dimensional space was accomplished. Use of three of these beacons helped accomplish determination of the orientation of a larger object, such as the simulated space station. The success of artificial neural networks in distinguishing between these beacons (in a Cartesian coordinate frame) was necessary to calculate the normal point projecting from the docking port and further automate the docking process. From this information,

waypoints and trajectories were calculated to the station. These valuable sensor technologies, coupled with other developing technologies such as artificial intelligence, will permit practical automation of space operations on a large scale.

With the incredible potential that automation technologies offer, scientists will definitely proceed with their studies in these fields. Machine vision is one of the major keys to the fascinating fields of robotics and automation.

Footnotes

[1] The Soviet Space Programme. R. Humble. Routledge Co. London and New York. 1988.

[2] The Soviet Year In Space 1988. N. Johnson. Teledyne Brown Engineering. Colorado Springs, Colorado. 1988.

[3] The Soviet Year In Space 1989. N. Johnson. Teledyne Brown Engineering. Colorado Springs, Colorado. 1989.

[4] Space Shuttle. M. Smith. Haynes Publications, Inc. Newbury Park, California. 1986.

[5] The Space Shuttle At Work. H. Allaway. Scientific and Technical Information Branch and Division of Public Affairs, NASA. Washington, D.C. 1979.

[6] "Application of Rapid Automatic Passive Optical Ranging (RAPOR) to Ship Control". Proceedings of the Ninth Ship Control Systems Symposium. vol. 4. pp. 4.426-4.437. W.I. Clement and K.A. Knowles. 1990.

[7] Robot Motion: Planning and Control. M. Brady et. al. The MIT Press. Cambridge, Massachusetts. 1984.

[8] A Practical Guide to Neural Nets. M. Nelson and W. Illingworth. Addison-Wesley Publishing Company. Reading, Massachusetts. 1991.

[9] "Visual Pattern Recognition by Moment Invariants". IRE Transactions on Information Theory. vol. IT-8, pp. 179-187. M. K. Hu. 1962.

Bibliography

Computing Depth from Temporal Cross-Correlation: A Comparison of Two Methods of Computation. J. A. Horst. National Institute of Standards and Technology (NEL), Gaithersburg, MD. Unmanned Systems Group.

Near Real-Time Stereo Vision System: Patent Application. L. H. Matthies and C. H. Anderson. National Aeronautics and Space Administration, Pasadena, CA. Pasadena Office.

Kinematic Calibration of an Active Camera System. G. S. Young, T. H. Hong, M. Herman, and J. C. S. Yang. National Institute of Standards and Technology (NEL).

Marker Recognition Using a Single Transputer. C. C. Chiu, R. G. Gosine, and R. D. Jackson. Cambridge University (England) Department of Engineering.

Real-Time Model-Based Tracking Combining Spatial and Temporal Features. K. Chaconas and M. Nashman. National Institute of Standards and Technology (NEL).

Recognition of Three Dimensional Objects Using HU-Invariants: Technical report. R. Lopez-Bonilla and B. Singh. Bradford University (England) Postgraduate School of Electrical and Electronic Engineering.

Range from Triangulation Using an Inverse Perspective Method to Determine Relative Camera Pose. K. Chaconas. National Institute of Standards and Technology.

Determining the Translation of a Rigidly Moving Surface, without Correspondence. Defense Technical Information Center. Rochester Computer Science Department. John Aloimonos.

Determining the 3-D Motion of a Rigid Surface Patch without Correspondence, under Perspective Projection: I. Planar Surfaces. II. Curved Surfaces. John Aloimonos and Isidore Rigoutsos. Rochester Computer Science Department.

Binocular Image Flows: Steps Toward Stereo-motion Fusion. Allen M. Waxman. University of Maryland Computer Vision Laboratory Center for Automation Research.

Stereo and Eye Movement. Davi Geiger and Alan Yuille. Massachusetts Institute of Technology Artificial Intelligence Laboratory.

Space Environment Robot Vision System. H.J. Wood and W.L. Eichhorn. NASA Goddard Space Flight Center, Greenbelt, MD.

Telepresence and Space Station Freedom Workstation Operations. D.G. Jensen and S.C. Adam. NASA Johnson Space Center, Houston, TX.

Depth Perception in Remote Stereoscopic Viewing Systems. NASA, Washington, D.C.

Connectionist Model-Based Stereo Vision for Telerobotics. W. Hoff and D. Mathis. Martin Marietta Aerospace, Denver, CO.

A Practical Guide to Neural Nets. M. Nelson and W. Illingworth. Addison-Wesley Publishing Company. Reading, Massachusetts. 1991.

Introductory Computer Vision and Image Processing. A. Low. McGraw-Hill Book Company. England. 1991.

The C Programming Language. B. Kernighan and D. Ritchie. Prentice Hall. Englewood Cliffs, New Jersey. 1988.

Introduction to Robotics. P. McKerrow. Addison Wesley Publishing Co. New York, New York. 1991.

Robot Motion: Planning and Control. M. Brady et. al. The MIT Press. Cambridge, Massachusetts. 1984.

Artificial Intelligence for Space Station Automation. O. Firschein et. al. Noyes Publications. Park Ridge, New Jersey. 1986.

The Space Station. K. Alexander. Gallery Books. New York, New York. 1988.

Space Shuttle. M. Smith. Haynes Publications, Inc. Newbury Park, California. 1986.

The Space Shuttle At Work. H. Allaway. Scientific and Technical Information Branch and Division of Public Affairs, NASA. Washington, D.C. 1979.

The Soviet Space Programme. R. Humble. Routledge Co. London and New York. 1988.

The Soviet Year In Space 1988. N. Johnson. Teledyne Brown Engineering. Colorado Springs, Colorado. 1988.

The Soviet Year In Space 1989. N. Johnson. Teledyne Brown Engineering. Colorado Springs, Colorado. 1989.

Space Technology. K. Gatland. Salamander Books Ltd. New York, New York. 1989.

Encyclopedia of Space. N. Booth. Mallard Press. New York, New York. 1990.

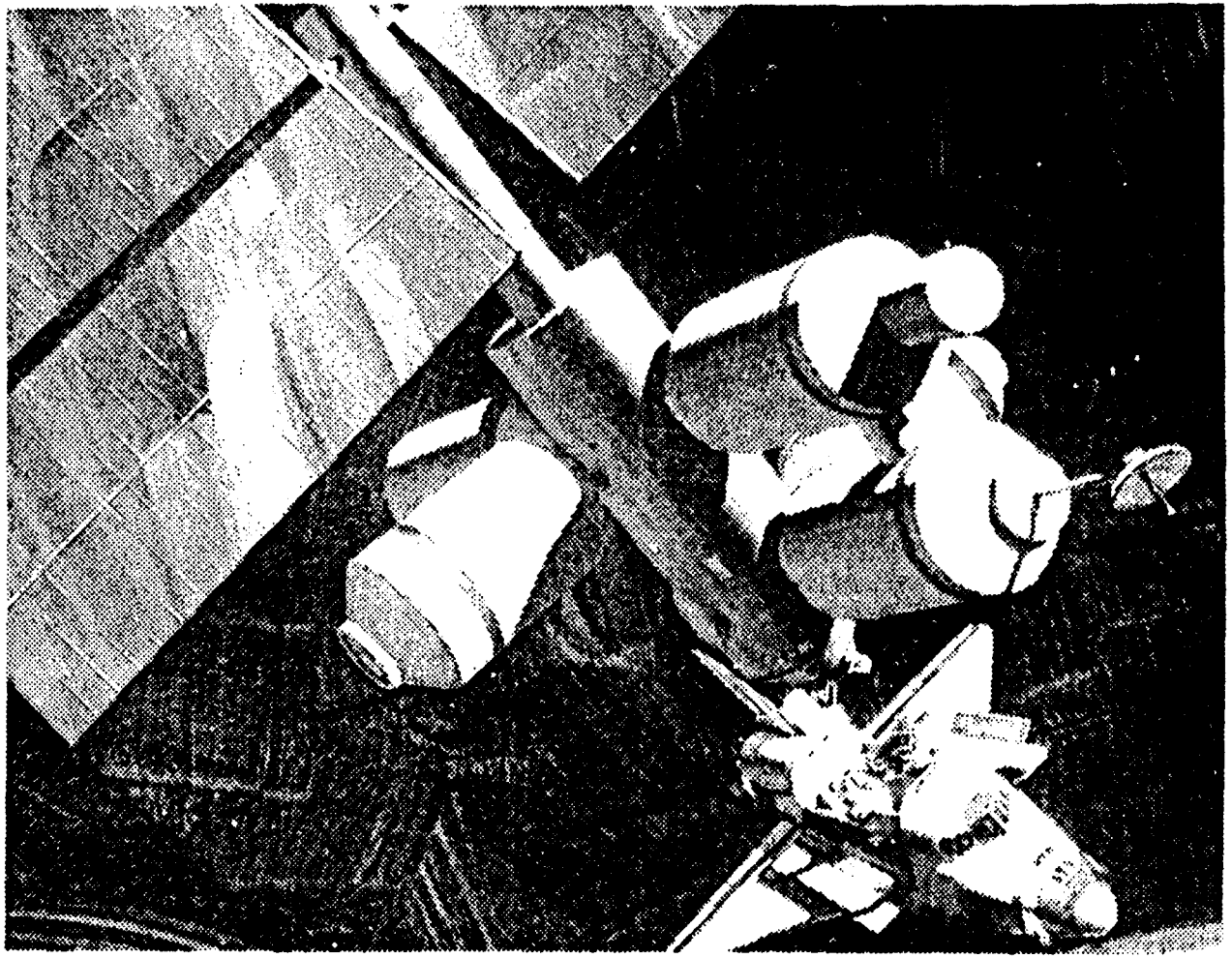
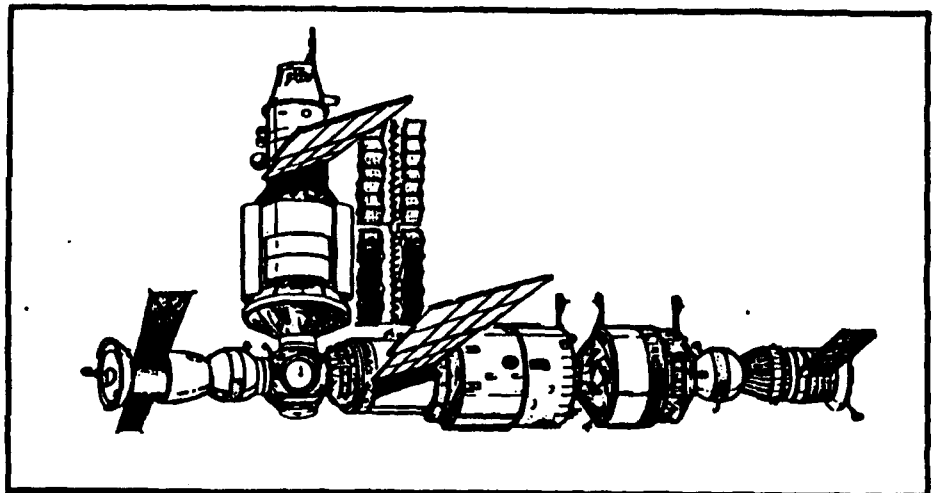


Figure 1: Shuttle (with Docking Adapter) about to rendezvous with futuristic space station.

(Taken from Space Station, 1988, K. Alexander)

TABLE 19. THE MIR SPACE STATION COMPLEX



COMPONENT	SOYUZ TM	KVANT 2	MIR	KVANT 1	PROGRESS M	TOTAL COMPLEX
YEAR OF DEBUT	1986	1989	1986	1987	1989	1989
LENGTH, m	7.8	12.4	13.1	5.8	8.8	34
MAXIMUM BODY DIAMETER, m	2.7	4.4	4.2	4.2	2.7	4.4
INITIAL MASS, metric tons	7.1	19.6	20.4	11	7.2	~ 70
USEFUL VOLUME, m^3	10	62	90	40	7.6	210
NUMBER OF SOLAR ARRAYS	2	2	3	--	2	9
SOLAR ARRAY SPAN, m	10.6	24	29.7	--	10.6	29.7
TOTAL AREA OF SOLAR ARRAY, m^2	10	90	90	--	10	174
MAXIMUM POWER GENERATION, KW	1.3	6.7	10.1	--	1.3	19.4[a]

[a] CURRENT POWER IS LESS DUE TO MIR SOLAR PANEL DEGRADATION

Figure 2: The Mir Space Station, its modules, and their specifications. (Taken from The Soviet Year in Space, 1989, N. Johnson)

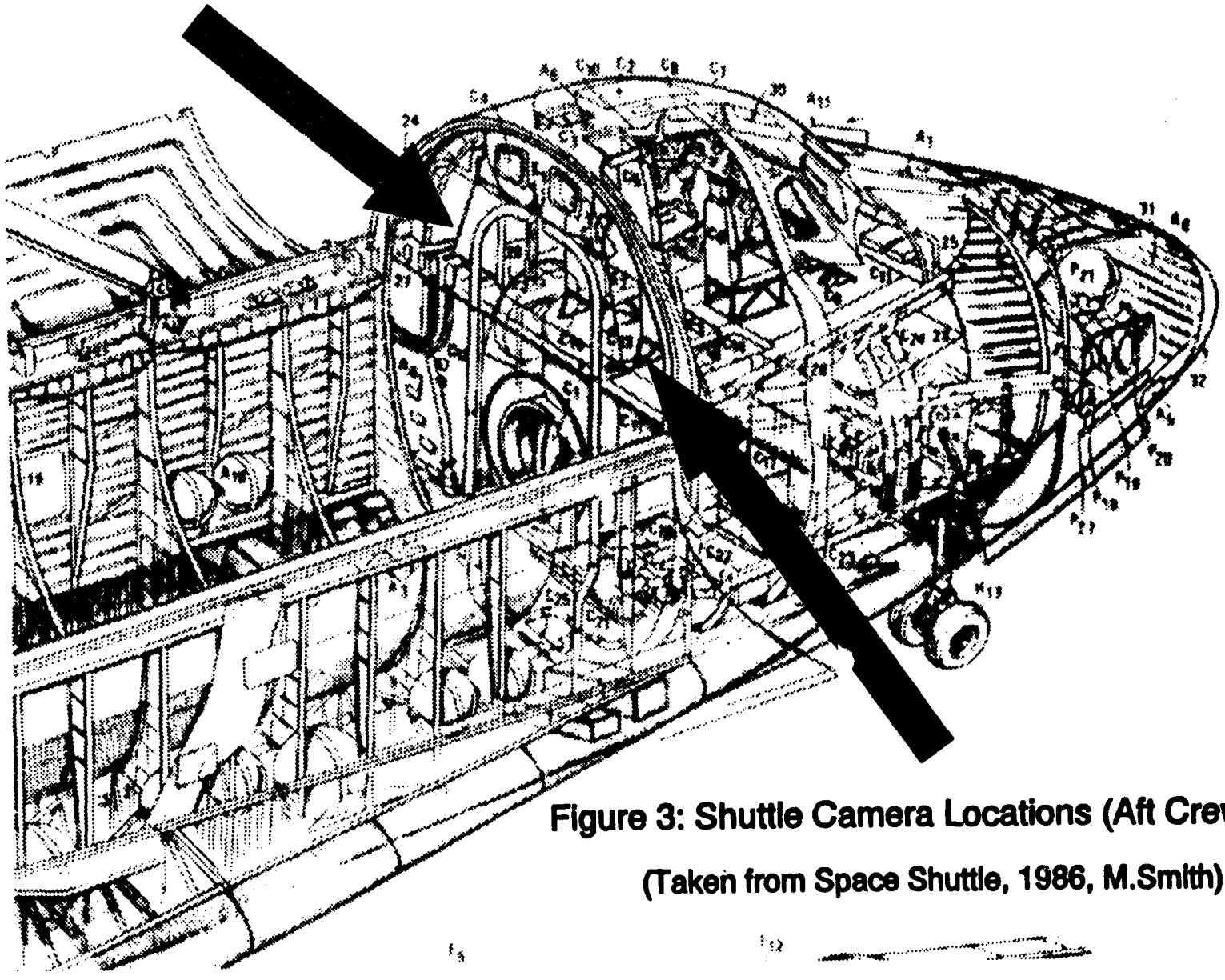


Figure 3: Shuttle Camera Locations (Aft Crew Station)

(Taken from Space Shuttle, 1986, M.Smith)

Hardware Configuration

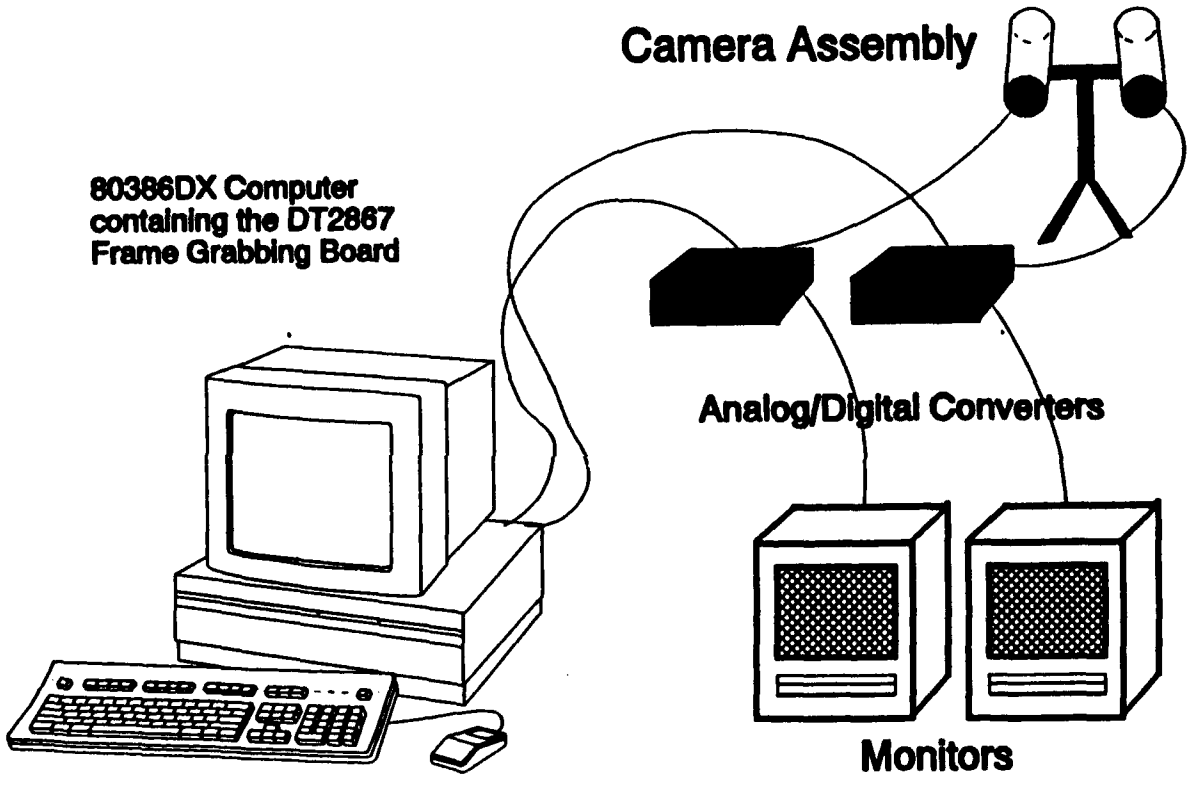


Figure 4: Hardware Configuration and Basic System Setup

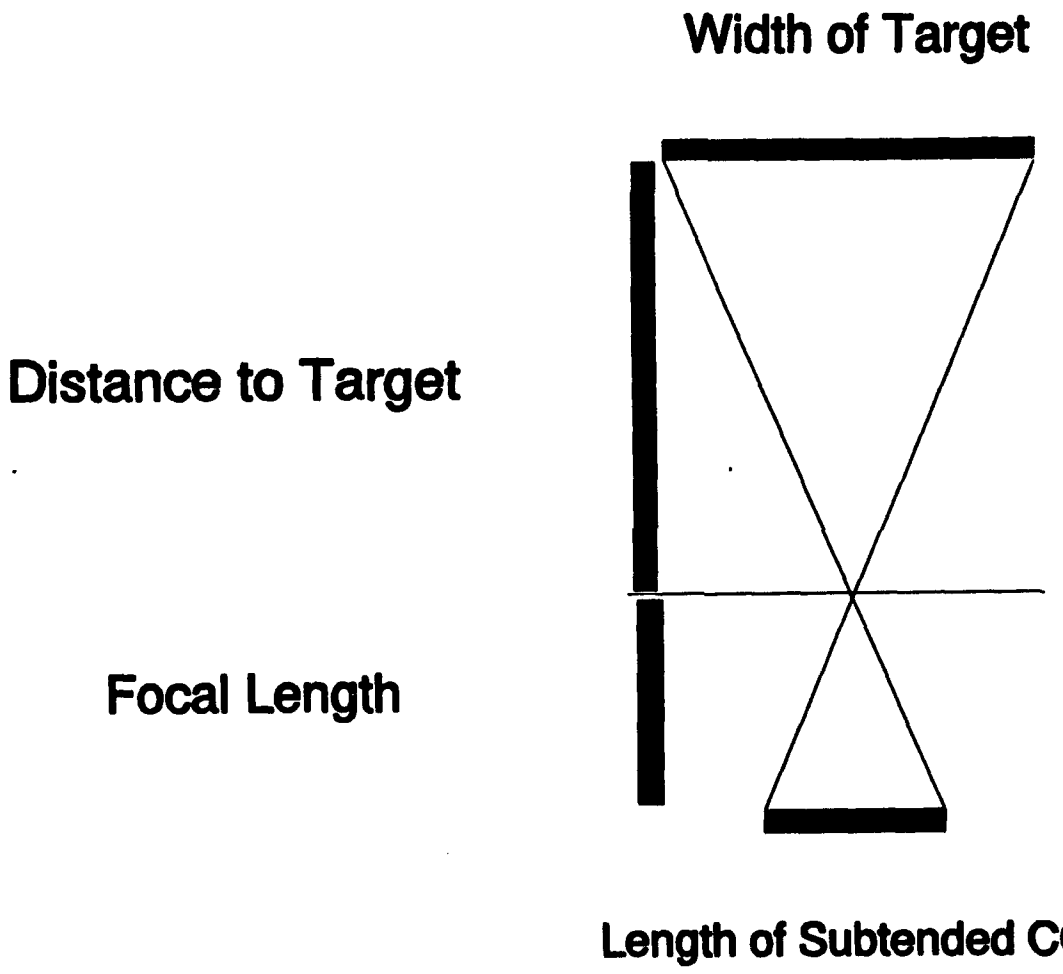


Figure 5: Relationship used to determine Pixels per inch

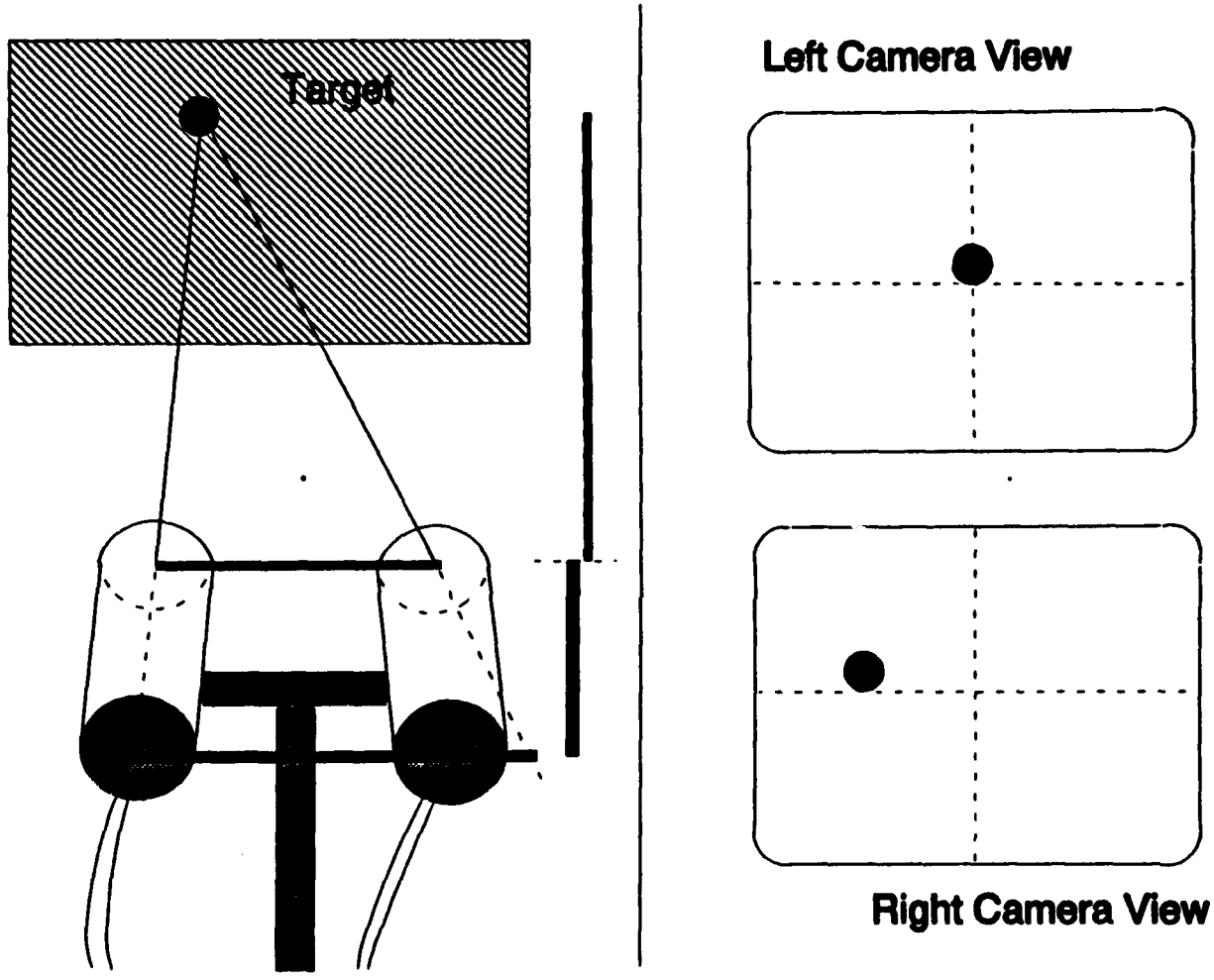
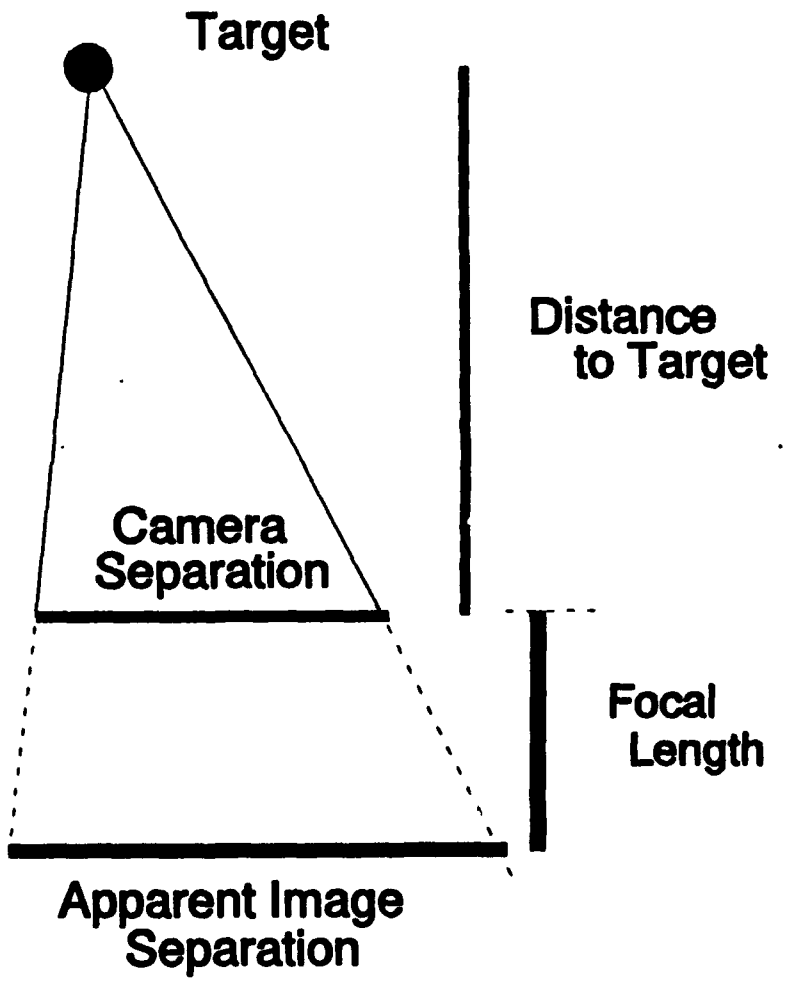


Figure 6: Stereo Vision - Camera Setup, Views, and Variables



$$\frac{\text{Distance to Target}}{\text{Camera Separation}} = \frac{\text{Distance to Target + Focal Length}}{\text{Apparent Image Separation}}$$

Apparent Image Separation =

Camera Separation +

(Image Parallax (in pixels) X

Pixels/Inch)

Figure 7: Stereo Vision Variables and Equations

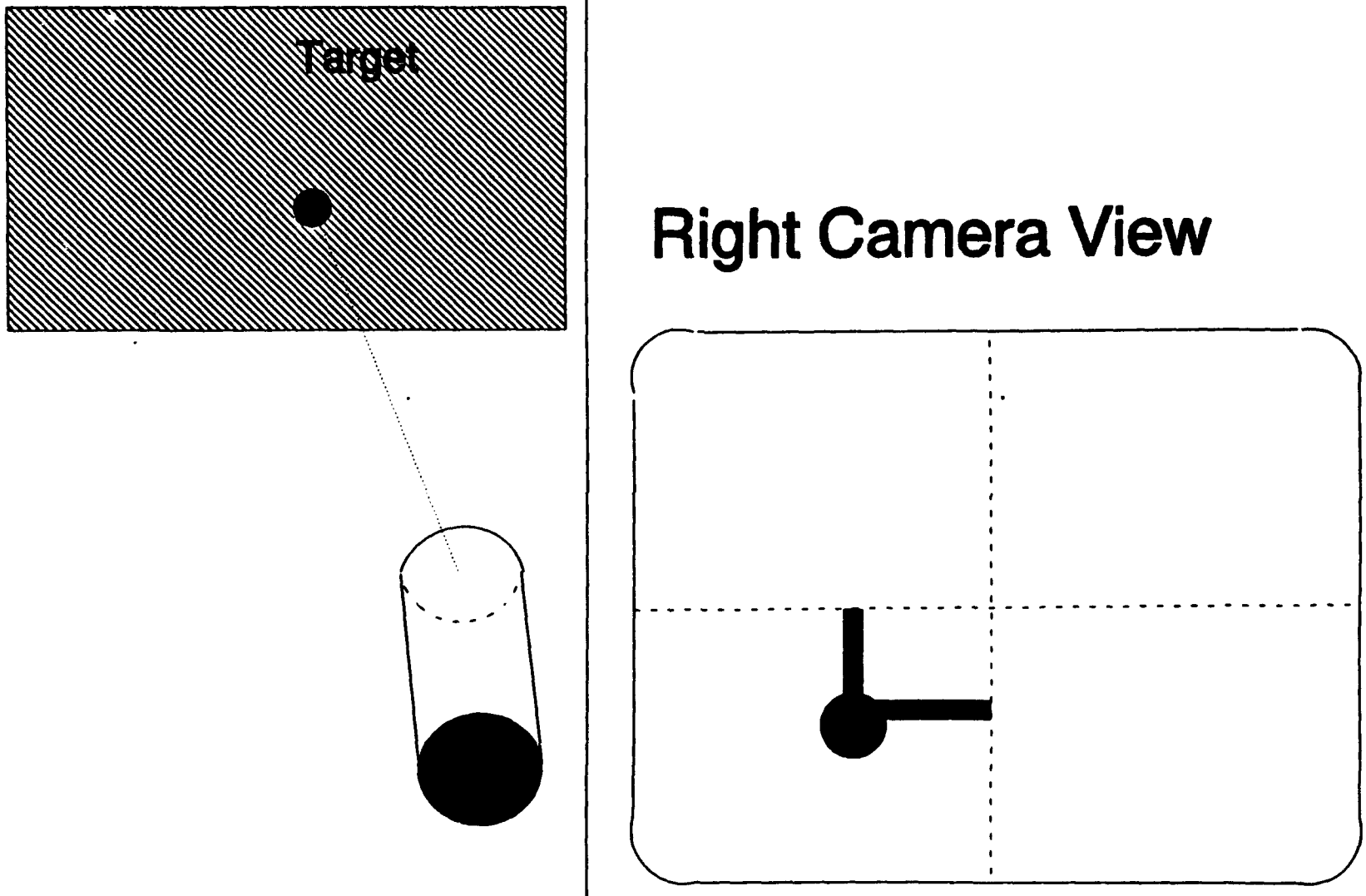
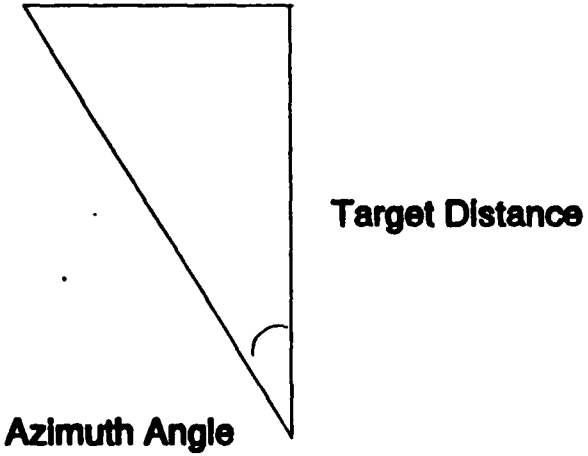


Figure 8: Bearing Determination Diagram and Camera View

X Offset

(X distance) X (Inches/Pixel)



Azimuth Angle =
ArcSin (X offset/
Target Distance)

Right Camera View

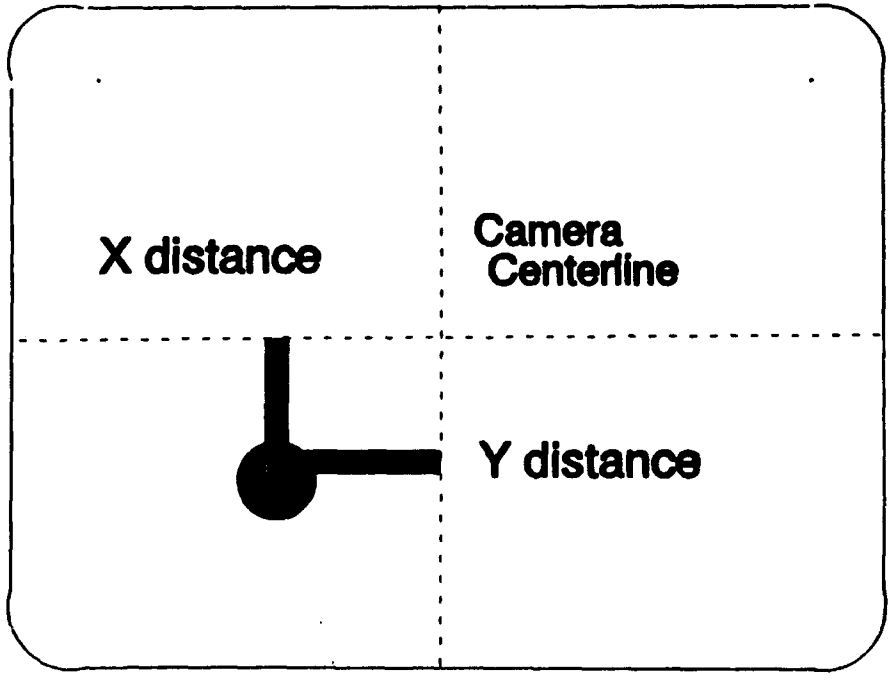


Figure 9: Bearing Determination Variables and Equations

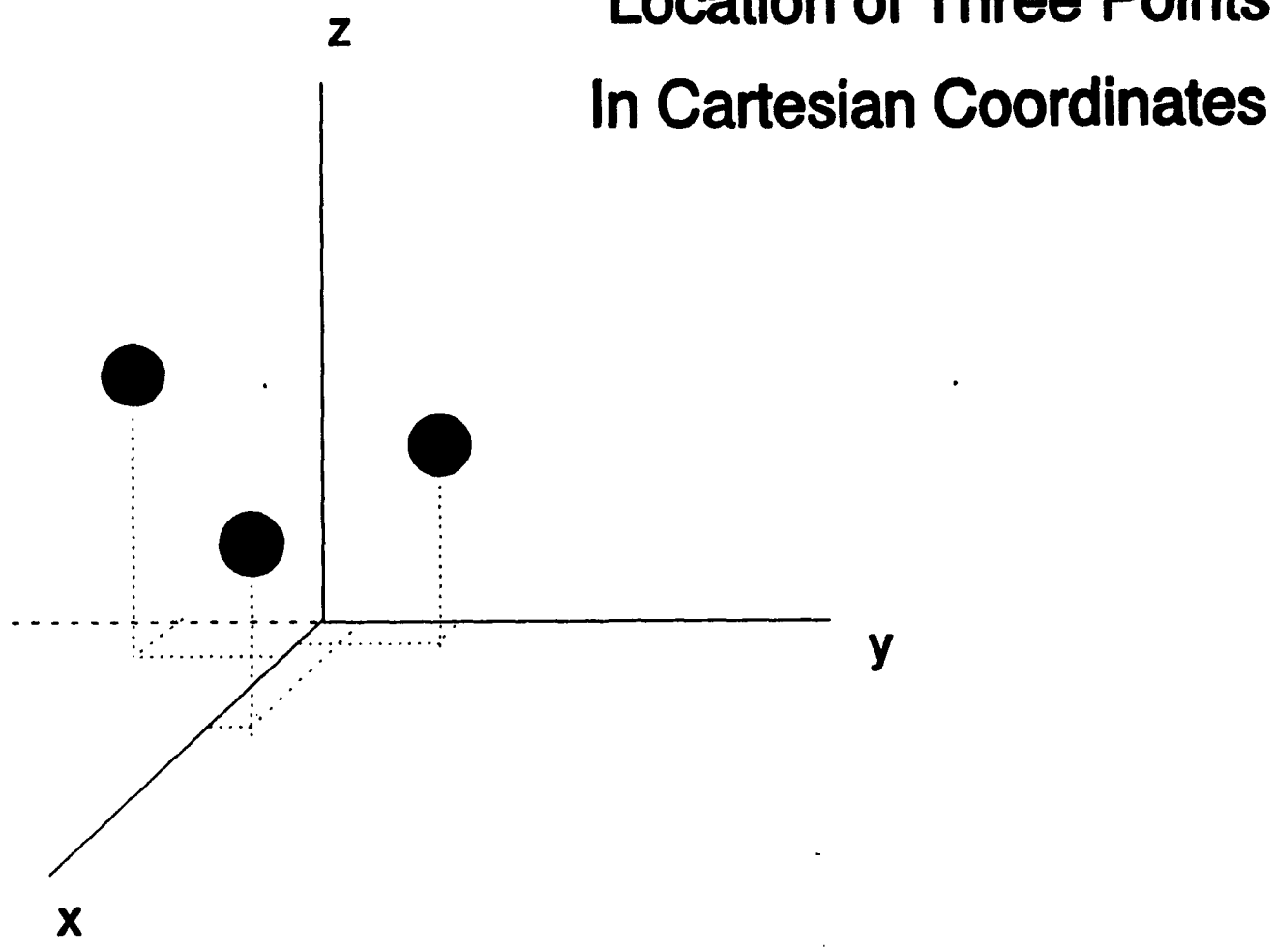


Figure 10: Graphic Model Depicting 3-D Loactions of Objects

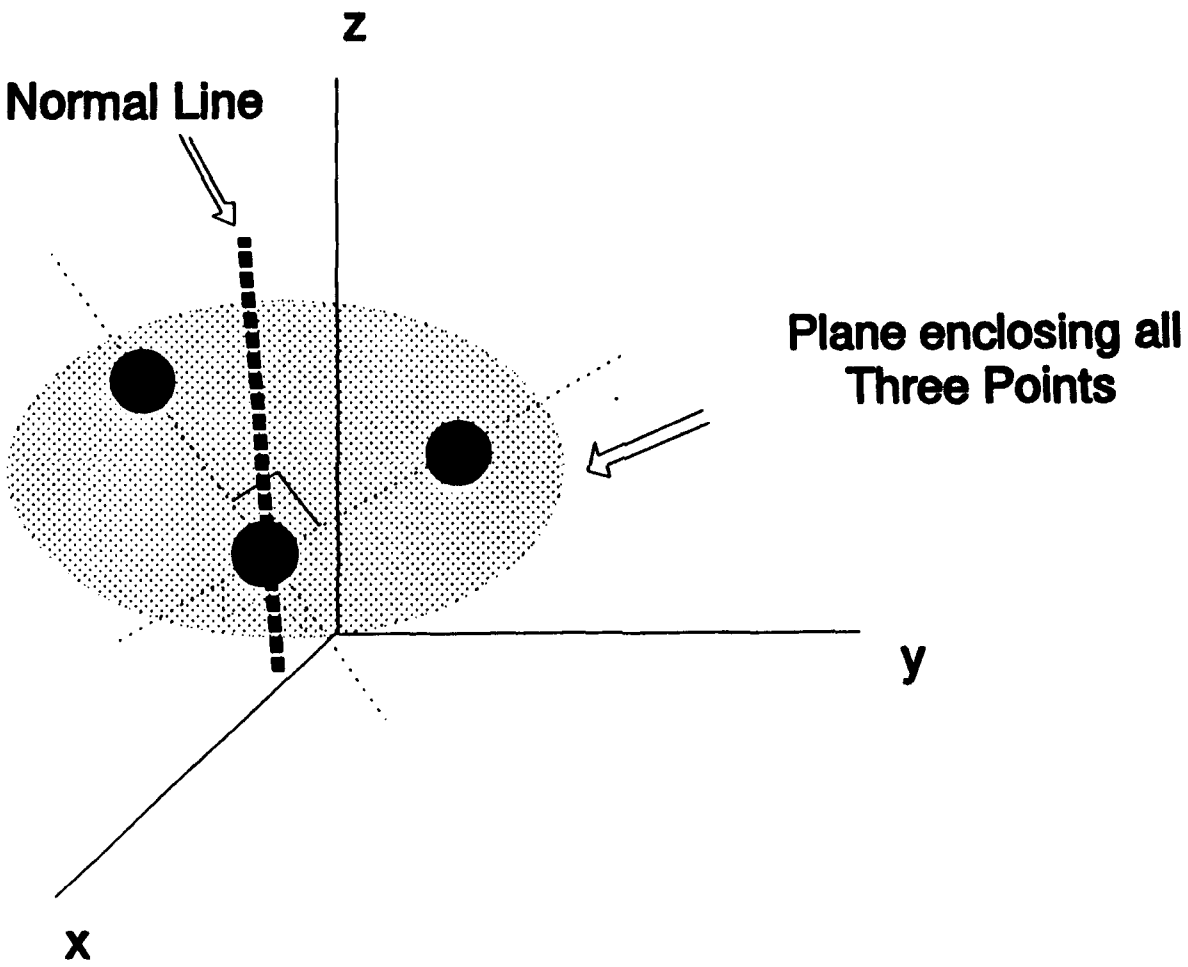


Figure 11: Depiction of Determination of Normal Line through Space Station

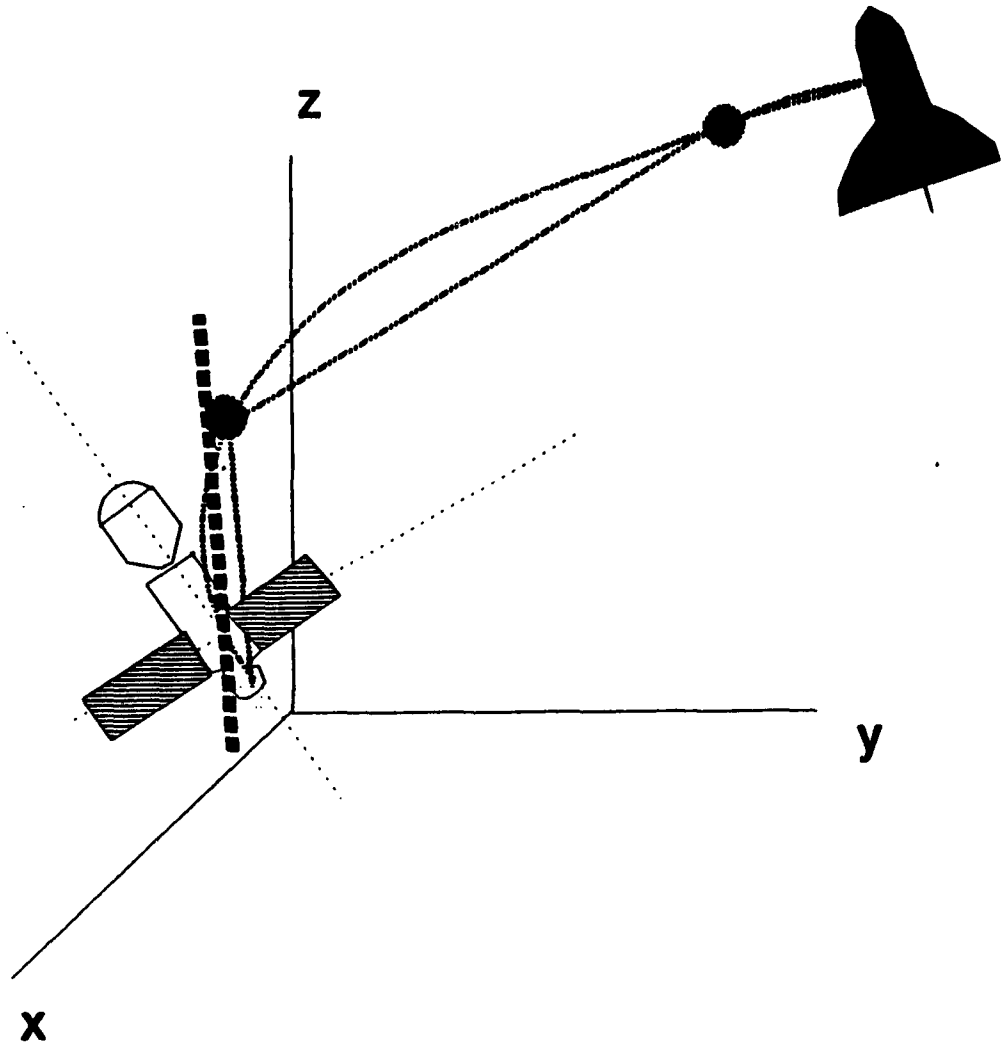


Figure 12: Depiction of Waypoint and Smooth Trajectory Determination

Mathematical Model of An Artificial Neuron

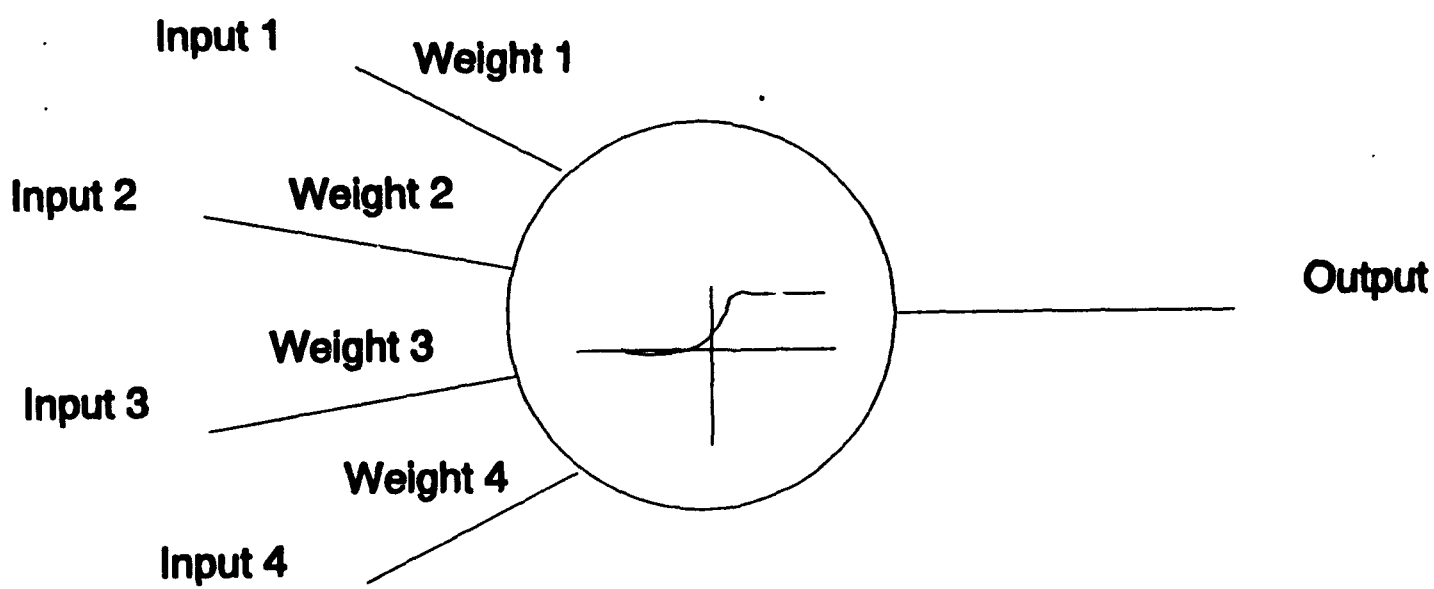


Figure 13: Mathematical Model of Artificial Neuron Operation

5 Input, 2 Output, Back-Propagation Artificial Neural Network

1 Hidden Layer Neuron

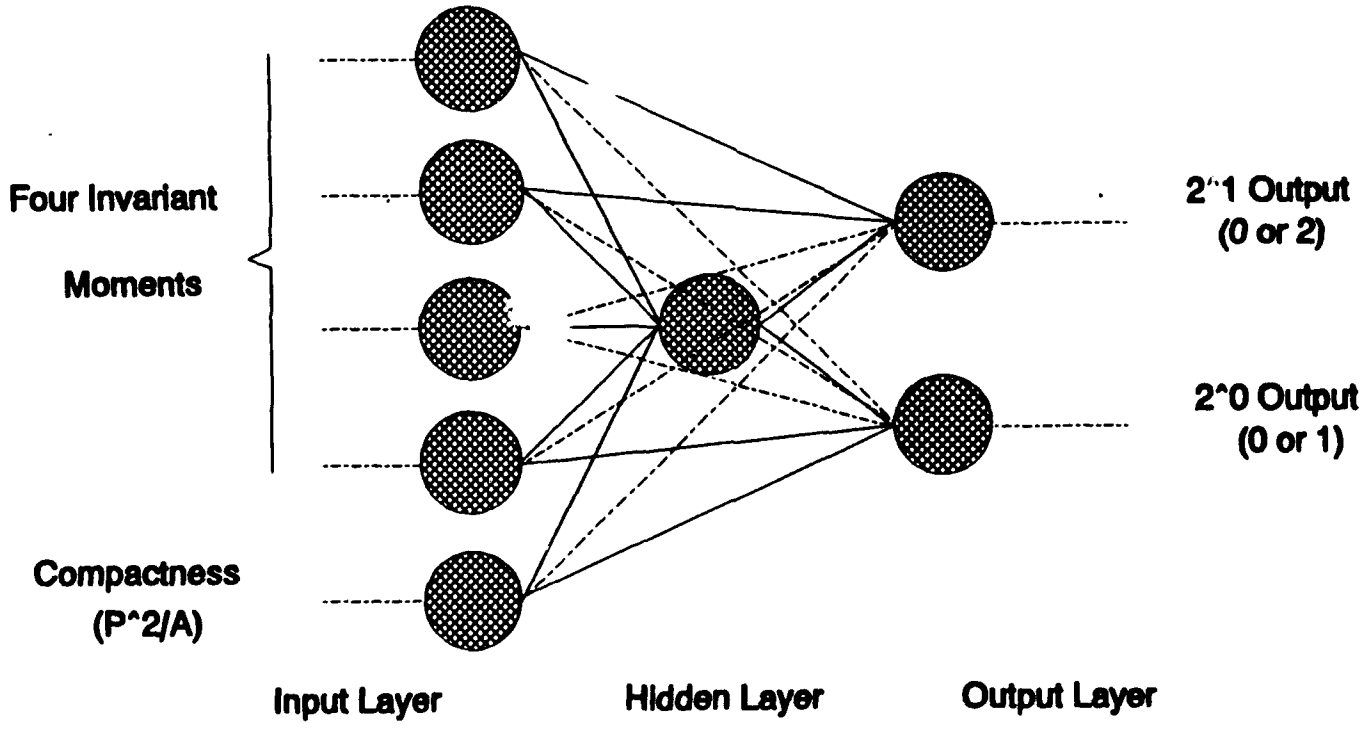


Figure 14: Model of Neural Network Used in Experiment

```
/* ---------------------------------------------- */
/* Stereoscopic Vision Routine written by Michael M. Hsu, MIDN 1/C 1993  */
/* This program uses the DT2867 Frame Grabbing board in conjunction with */
/* a pair of Panasonic mini-cameras and analog-to-digital converters.     */
/* Several routines were written with the assistance of Professor         */
/* William Clement of the Weapons and Systems Engineering Department      */
/* ---------------------------------------------- */

/* Run on Microsoft C 7.0, using 80286 processor, large memory model,  */
/* and 14336 stack options.  Include CELIB.LIB, DTIFMS.LIB, GRAPHICS.LIB */
/* and LIB67.LIB in the file list.                                       */


#include <conio.h>              /* for kbhit */
#include <malloc.h>
#include <dos.h>                /* required for Microsoft's open/close */
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <errno.h>
#include <stdio.h>              /* required for printf */
#include <stdlib.h>
#include <math.h>
#include <graph.h>
#include <dttyp.h>              /* file of DT specific types */
#include "c:\dt\include\ioctl.h"                        /* IOCTL stuff */
#include "c:\dt\include\ioctlext.h"
#include <cmd_eng.h>            /* command engine stuff */
#include <dt2867\reg_num.h>     /* register numbers */
#include <dt2867\lib67.h>       /* dt2967 library interface functions */
#include <id_info.h>            /* needed by IOCTL commands */
#include <string.h>             /* for chain code string operations */

/* mouse routine includes */
#include <dos.h>                                        /* for int86(), int86x() */
#include "mouse.h"                                      /* for "struct cursor_struct" */


struct object {         /* this structure holds the information for              */
   int row;                     /* objects analyzed by the chain_code function      */
   int col;
   float xbar;          /* row and col designate the location of the             */
   float ybar;          /* starting pixel from which chain_code operates.*/
   float phi[5];        /* xbar and ybar designate the centroid, phi[5]  */
   float compactness;           /* designates the first 4 invariant moments, and */
   };                           /* compactness returns the compactness of the    */
                                /* object analyzed                               */

struct beacon {
   float xbar[2],ybar[2];       /* camera 0 and 1 centroids of beacon */
   float xdist,ydist,zdist;     /* cartesian coordinates of beacon */
   };

struct waypoint {
   float matrix[4][4];          /* homogeneous transform matrix at this  */
   float xdist,ydist,zdist;     /* waypoint. also, the coordinates and   */
   float pitch,yaw,roll;        /* orientation of the shuttle here.      */
   };

struct trajectory {
   float time;                  /* individual trajectory time, position, */
   float xdist[2],ydist[2],zdist[2];            /* velocity, and acceleration at */
   float pitch[2],yaw[2],roll[2];  /* this point in the trajectory. */
   };

struct delta {                          /* for purposes of trajectory smoothing, */
```

```
    float xdist,ydist,zdist;    /* the deltab and deltac differences in */
    float pitch,yaw,roll;       /* position and orientation between     */
    };                                  /* three successive waypoints           */

struct point2 {                 /* generic structure format for a       */
    int matrix[2];              /* 2-point vector                       */
    };

struct point3 {                 /* generic structure format for a       */
    float matrix[3];            /* 3-point vector                       */
    };

struct homog {                          /* generic structure format for a       */
    float matrix[3][3];         /* 3-by-3 homogeneous matrix            */
    };

        char            cont;               /* continue flag, 0 to repeat   */
        short int   color;          /* short integer color index    */
        int             polarity;   /* flag to indicate background/object contrast direction */
        int             numbeacon;          /* counter for cycling through beacons */
        int             numwaypoint;        /* counter for cycling through waypoints */
        float           ccenter[2];         /* centerline columns for cameras 0 and 1 */
        float           rcenter[2];         /* centerline rows for cameras 0 and 1 */
        int             classnumber;        /* number of class beacon is recognized as */

        /* for computation of the orientation of a three-beacon structure */
        float           xx,xy,xz;   /* x unit vector */  /* waypoint homogeneous matrix */
        float           yx,yy,yz;   /* y unit vector */  /* rotation values */
        float           zx,zy,zz;   /* z unit vector */
        float           xd,yd,zd;   /* 'unit' vector magnitudes for scaling purposes */

        float           tacc, tau, h, tleg, vmax, distance, tvmax;                  /* tacc */

        /* for transformation between coordinate systems - normal and world */
        float           vector1[4];         /* input vector of change in distance along normal */
        float           vector2[4];         /* output vector of actual distances along axes */

        float           view_position[3];

        /* neural network classification constants */
        float           data_in[5];         /* 5 input values for network */
        float           data_out[2];        /* 2 output values for network */
        float           result;     /* weighted sum of outputs for classification */

        /* structure allocations */
        struct object       object2;            /* object sent to chain_code */
        struct beacon       beacons[3];                 /* 3 beacons identified and analyzed */
        struct waypoint waypoints[5];           /* 5 waypoints computed for docking */
        struct trajectory   trajectories[40];   /* 40 trajectory points analyzed */
        struct delta        deltab, deltac, a;  /* difference computation between waypoints */

        /* for draw_all_figures and graphics routines */
        struct point3       pts[4];     /* for drawing coordinate axes */
        struct point3       p3a;
        struct point3       p3b;

        struct point2       p2a;
        struct point2       p2b;

        struct homog        AA[3];

        int             maxX,maxY;          /* max pixel count for graphics mode    */
        unsigned char       *axis_text[4];      /* for marking identity of axes  */
        unsigned char       fillmask[8];        /* for later filling in of polygons     */
        float           aspect;     /* aspect ratio of monitor              */
        float           scale;              /* scaling of image                     */
        float           dist;               /* distance of 'eye' from origin */
```

```
float           xscale, xoffset;
float           yscale, yoffset;
float           view_theta;
float           view_gamma;
float           view_theta_home;
float           view_gamma_home;
float           angles[3];          /* vector of angles */

/* for translation from trajectory to plotting points */
struct point3   path[40];  /* structure format usable by transform_point */
struct point2   pathplot[40];       /* output format of trajectory from transform_point */


/* for mouse routines */
union     REGS    regs;
struct    SREGS   sregs;


/*        for determining when to change cursor shape:  */
static int  MAX_x, MAX_y;           /* number of pixels in each directions */
static int  b_top=75, b_bottom=75, b_left=100, b_right=100;
static int  u_flag=1, l_flag=1;

/*        general mouse variables:  */
static int  status, num_buttons;
static int  driver_installed_flag=0;

static unsigned             int_seg, int_off, int_mask;



/* information gathering routines for stereo vision */
void take_pix(int,int,int);
void take_two_pix (int);
void get_point(int *,int *,int,int);
void chain_code(struct object *object1, float alpha,int threshold,int buff,int fd);

/* graphical trajectory display routines */
void init();
void compute_view_transform(float theta, float gamma, float matrix[3][3]);
void compute_RPY_transform(float yaw, float pitch, float roll, float matrix[3][3]);
void compute_AzEl_transform(fl  az, float el, float matrix[3][3]);
void transform_point(int tr, float p1[3], int p0[2]);
void draw_all_figures();
void display_trajectory();

/* neural network classification routine */
void classify();
/* ------------------------------------------------------------ */
int main()
{


        int             i,j,k;              /* loop counters */

        /* mouse routine variables */
        int             x, y;               /* old mouse position */
        int             x1, y1;     /* new mouse position */
        int             p1;                 /* button press/release indicator */
        int             p2, p3;     /* button press location */
        int             p4, p5;     /* button release location */
        int             mstatus;  /* mouse status indicator */
        int             flag;               /* flag for mouse button press */
        int             ok_flag=0;          /* flag for restoration of image */

        /* get_point to chain_code transport variables */
```

```
        int             colnum,rownum;          /* values obtained from mouse routine */

        /* monitor cursor variables */
        int             row, colt1, colrt1;     /* row, left column, right column numbers */
        int             rowup2,rowdn2,col;      /* col, top row, bottom row numbers */

        /* camera and ILUT information for DT2867 board */
        int             buff, camera;           /* buffer/camera numbers */
        int             fd, status;             /* DT2867 board handle and status indicator */
        int             thresh, threshold;      /* binarization and comparison thresholds */

        int             cam0num, cam1num;       /* number of pixels in camera 0,1 line arrays */
        int             camera0[768];           /* camera 0 line array */
        int             camera1[768];           /* camera 1 line array */

        /* variables for distance computation using edge detection along rows - not used */
        int             found, pixbegin;        /* edge found flag, starting pixel value */
        int             edgecol0,edgecol1;      /* object edge in camera 0,1 */

        float           time;
        float           alpha;

        /* stereoscopic ranging constants */
        float           scalfact;               /* scaling factor in millimeters */
        float           focaleng;               /* focal length in millimeters */
        float           camsep;                 /* camera separation in millimeters      */

        /* stereoscopic ranging variables */
        int             pixdiff;                /* difference in pixels */
        float           parallax;               /* parallax between two camera images */
        float           distance;               /* computed distance from stereo vision */

        u_long          red[256], green[256], blue[256];
        XY_rgn_buf      rgn, rgn2, blank_line, blank_line2;

/* START OF MAIN PROGRAM */

  /* set video mode to 16-color VGA */
  _setvideomode( _VRES16COLOR );

  /* clear the screen */
  _clearscreen( _GCLEARSCREEN);

  /* open the DT2867 board */
  if ( (fd=open("DT2867#0", O_RDWR))==-1 ) {
        printf("Error opening DT-2867 device.  Exiting ...\n\n");
        exit(1);
        }

  dt67_disp (fd, DT67_ON);                  /* turns the device on */
  dt67_inp_sync (fd, DT67_SYNC_CUR);              /* use current channel as sync source */
  dt67_inp_timing (fd, DT67_EXTERNAL);   /* timing source is external to board */

  /* initial configuration of Input Look Up Tables */
  for (i=0; i<256; i++)          /* Set ILUT values */
        green[i]=i;                         /* Normal picture */
  status = dt67_inp_lut (fd, DT67_WRITE, DT67_ILUT, DT67_LUT0, 0, 256, green);
  if (status != E_NORMAL)
        printf("Abnormal ILUT operation ...\n");

/*  Un-comment this function for demonstration of two-camera parallax */
/*  take_two_pix(fd); */


/*  This section simply takes a single picture with camera 0, allows the user
    to designate an object, and find its centroid using chain_code */
```

```
/* Take first picture using camera 0 */
/* printf("\nPreparing to take picture with camera 0:");
camera=0;
buff=0;
take_pix(camera,buff,fd);

buff=0;
get_point(&colnum,&rownum,buff,fd);
printf("\n(col %3d,row %3d)  is the position selected ",colnum,rownum);
object2.row=rownum;
object2.col=colnum;
alpha=1.0;
threshold=128;
chain_code(&object2,alpha,threshold,buff,fd);
printf("\n(%6f,%6f) is the (row,col) of the centroid",object2.xbar,object2.ybar);
alpha=1.0; */



alpha=1.0;              /* camera pixels are square */
threshold=128;  /* middle grey-value to allow detection of 0 and 255 */

/* determine centerline pixels of cameras */
for(camera=0;camera<=1;camera++) {
        rcenter[camera]=255.5;                  /* 512/2-.5 */
        ccenter[camera]=383.5;                  /* 768/2-.5 */
        }


/* This loop cycles through both cameras,              */
/* taking pictures of and analyzing each beacon */
for(camera=0;camera<=1;camera++) {
        buff=camera;                            /* buffer corresponding to camera */
        take_pix(camera,buff,fd);       /* take picture and threshold */

        /* This loop cycles through all three beacons in the current image */
        for(numbeacon=0;numbeacon<=2;numbeacon++) {
           printf("\n\nCamera #%3d, Buffer #%3d, Beacon #%3d ",camera,camera,numbeacon);
           printf("\nPlease select a point to the left of beacon %3d ",numbeacon);
           get_point(&colnum,&rownum,buff,fd);          /* get start row and column from mouse */
           object2.row=rownum;                          /* give object2 structure mouse inputs */
           object2.col=colnum;
           polarity=0;                                  /* black on white polarity */
           chain_code(&object2,alpha,threshold,buff,fd);   /* perform chain code on beacon */
           beacons[numbeacon].xbar[camera]=object2.xbar;  /* give beacons structure centroid values */
           beacons[numbeacon].ybar[camera]=object2.ybar;
           }
        }

        /* Beacon identification routine using neural networks */
        /* take picture with camera 0 */
        camera = 0;
        buff = camera;
        take_pix(camera,buff,fd);

        for(j=0;j<=2;j++) {
           printf("Camera #%3d, Buffer #%3d, Object #%3d ",camera,camera,j);
           printf("\nPlease select a point to the left of object %3d ",j);

           /* mouse input to left of graphical designator */
           get_point(&colnum,&rownum,buff,fd);
           object2.row=rownum;
           object2.col=colnum;

           /* chain coding of white on black object */
           polarity=1;
           chain_code(&object2,alpha,threshold,buff,fd);
```

```
/* returned descriptors for use by neural network */
data_in[0]=object2.phi[1];
data_in[1]=object2.phi[2];
data_in[2]=object2.phi[3];
data_in[3]=object2.phi[4];
data_in[4]=object2.compactness;
for (i=0;i<5;i++)
        printf("\n%6f",data_in[i]);

/* neural network classification of object using descriptors */
classify();
printf("\nNeural Net Output #1 = %6f, Output #2 = %6f",data_out[0],data_out[1]);

/* interpretation of network output */
result=2*data_out[0]+data_out[1];
if (result<=.5)
        classnumber=0;
else
        if ((result>.5) && (result<=1.5))
           classnumber=1;
        else
           if ((result>1.5) && (result<=2.5))
                classnumber=2;
           else
                if ((result>2.5) && (result<=3.5))
                   classnumber=3;
                else
                   classnumber=100;
printf("\nObject Belongs to Class Number: %3d",classnumber);
}


/* Calibration of camera centerline pixels, pitch, yaw, roll? */
/* knowledge of the camera lens optics would help increase accuracy */

/* Coordinate Computation Routine */
/* Step 1: determination of beacon distances from cameras */
scalfact = (1.0/2555.0)*25.4;                   /* scaling factor in millimeters per pixel */
focaleng = 7.5;                                         /* focal length in millimeters */
camsep = 4.0*25.4;                              /* camera separation in millimeters */
for(numbeacon=0;numbeacon<=2;numbeacon++) {
   pixdiff = beacons[numbeacon].ybar[0]-beacons[numbeacon].ybar[1];
   printf("\n\nPixel difference between images = %4d pixels",pixdiff);
   parallax = pixdiff * scalfact;               /* parallax in millimeters */
   distance = focaleng * camsep / parallax;     /* distance from cameras in millimeters */
   printf("\n%6f",parallax);
   printf("\nDistance from cameras in millimeters = %6f",distance);
   printf("\nDistance from cameras in inches = %6f",distance/25.4);

   /* z distance in millimeters - straight out from shuttle */
   beacons[numbeacon].zdist = distance;

   /* from the camera's view, xdist is positive to the top along rows
                                   ydist is positive to the right along columns */

   /* determination of 'bearing' of beacons from camera setup */
   /* determine xdist and ydist from zdist, rcenter, ccenter, xbar, ybar */
   /* using only camera 0, and figuring in the proper ydist compensation */
   camera=0;
   pixdiff=rcenter[camera]-beacons[numbeacon].xbar[camera];
   beacons[numbeacon].xdist=beacons[numbeacon].zdist*pixdiff*scalfact/focaleng;
   pixdiff=beacons[numbeacon].ybar[camera]-ccenter[camera];
   beacons[numbeacon].ydist=(beacons[numbeacon].zdist*pixdiff*scalfact/focaleng)-(camsep/2);

   printf("\n xdist=%6f, ydist=%6f, zdist=%6f "
            ,beacons[numbeacon].xdist,beacons[numbeacon].ydist
            ,beacons[numbeacon].zdist);
```

```
    }
cont=getch();

/*      normalize vectors x (1->0), y (1->2) and take cross product to  */
/*      find vector along normal. Take the negative of this vector to   */
/*      get the z vector. Place these three vectors into a 3*3          */
/*      orientation matrix. Augment it with a position vector to        */
/*      get a 4*4 transform matrix. Note: this is a left-handed         */
/*      coordinate frame                                                */

/* x unit vector */
/* compute directional differences */
xx=beacons[0].xdist-beacons[1].xdist;
xy=beacons[0].ydist-beacons[1].ydist;
xz=beacons[0].zdist-beacons[1].zdist;
/* compute total magnitude of vector */
xd=sqrt(pow(xx,2)+pow(xy,2)+pow(xz,2));
/* normalize current vector into unit vector */
xx=xx/xd;
xy=xy/xd;
xz=xz/xd;

/* y unit vector */
/* compute in same manner as x unit vector */
yx=beacons[2].xdist-beacons[1].xdist;
yy=beacons[2].ydist-beacons[1].ydist;
yz=beacons[2].zdist-beacons[1].zdist;
yd=sqrt(pow(yx,2)+pow(yy,2)+pow(yz,2));
yx=yx/yd;
yy=yy/yd;
yz=yz/yd;

/* z unit vector */
/* use cross product of x and y, then use same method */
zx=xy*yz-xz*yy;
zy=xz*yx-xx*yz;
zz=xx*yy-xy*yx;
zd=sqrt(pow(zx,2)+pow(zy,2)+pow(zz,2));
zx=zx/zd;
zy=zy/zd;
zz=zz/zd;


vmax=25;            /* maximum velocity is 50 mm/sec */
tacc=10;            /* minimum time of acceleration/deceleration is 10 seconds */


/* formation of final waypoint matrix - docking point */
/* orientation matrix */
waypoints[4].matrix[0][0]=xx;
waypoints[4].matrix[1][0]=xy;
waypoints[4].matrix[2][0]=xz;
waypoints[4].matrix[0][1]=yx;
waypoints[4].matrix[1][1]=yy;
waypoints[4].matrix[2][1]=yz;
waypoints[4].matrix[0][2]=zx;
waypoints[4].matrix[1][2]=zy;
waypoints[4].matrix[2][2]=zz;
/* bottom row and scale factor */
waypoints[4].matrix[3][0]=0;
waypoints[4].matrix[3][1]=0;
waypoints[4].matrix[3][2]=0;
waypoints[4].matrix[3][3]=1;
/* position augmentation vector */
waypoints[4].matrix[0][3]=beacons[1].xdist;
waypoints[4].matrix[1][3]=beacons[1].ydist;
```

```
waypoints[4].matrix[2][3]=beacons[1].zdist;

/* Duplicate this point on waypoint before to stop at docking */
for(i=0;i<4;i++) {
   for(j=0;j<4;j++) {
            waypoints[3].matrix[i][j]=waypoints[4].matrix[i][j];
            }
   }

/* Duplicate this point and subtract v*tau from the z-axis */
/* this will allow a trajectory straight into the docking port */
for(i=0;i<4;i++) {
   for(j=0;j<4;j++) {
            waypoints[2].matrix[i][j]=waypoints[3].matrix[i][j];
            }
   }

/* compute components of negative normal as waypoint 2 and add in */
vector1[0]=0;
vector1[1]=0;
vector1[2]=-2*vmax*taoc;        /* this distance out from the station */
vector1[3]=1;
for(i=0;i<4;i++) {
   for(j=0;j<4;j++) {
            vector2[i]=vector2[i]+waypoints[2].matrix[i][j]*vector1[j];
            }
   }
for(j=0;j<4;j++)
   waypoints[2].matrix[j][3]=vector2[j];

/* initial waypoint is 0,0,0 RPY, and 0 translation */
/* orientation matrix */
waypoints[0].matrix[0][0]=1;
waypoints[0].matrix[1][0]=0;
waypoints[0].matrix[2][0]=0;
waypoints[0].matrix[0][1]=0;
waypoints[0].matrix[1][1]=1;
waypoints[0].matrix[2][1]=0;
waypoints[0].matrix[0][2]=0;
waypoints[0].matrix[1][2]=0;
waypoints[0].matrix[2][2]=1;
/* bottom row and scale factor */
waypoints[0].matrix[3][0]=0;
waypoints[0].matrix[3][1]=0;
waypoints[0].matrix[3][2]=0;
waypoints[0].matrix[3][3]=1;
/* position augmentation vector */
waypoints[0].matrix[0][3]=0;
waypoints[0].matrix[1][3]=0;
waypoints[0].matrix[2][3]=0;

/* second waypoint is some distance straight ahead, to cease velocity */
/* Duplicate this point and add a z-component ahead */
for(i=0;i<4;i++) {
   for(j=0;j<4;j++) {
            waypoints[1].matrix[i][j]=waypoints[0].matrix[i][j];
            }
   }
waypoints[1].matrix[2][3]=3*vmax*taoc;       /* this distance ahead */

/* computation of angles from orientation matrices */
for(k=0;k<5;k++) {
   waypoints[k].xdist=waypoints[k].matrix[0][3];
   waypoints[k].ydist=waypoints[k].matrix[1][3];
   waypoints[k].zdist=waypoints[k].matrix[2][3];
   waypoints[k].pitch=asin(-waypoints[k].matrix[2][0]);
   waypoints[k].yaw=asin(waypoints[k].matrix[2][1]/cos(waypoints[k].pitch));
```

```
    if ((waypoints[k].matrix[1][0]/cos(waypoints[k].pitch))>0) {
            if ((waypoints[k].matrix[0][0]/cos(waypoints[k].pitch))>0) {
               waypoints[k].roll=fabs(asin(waypoints[k].matrix[1][0]/cos(waypoints[k].pitch)));
               }
            else {
               waypoints[k].roll=acos(-1)-fabs(asin(waypoints[k].matrix[1][0]/cos(waypoints[k].pitch)));
               }
            }
    else {
            if ((waypoints[k].matrix[0][0]/cos(waypoints[k].pitch))>0) {
               waypoints[k].roll=-fabs(asin(waypoints[k].matrix[1][0]/cos(waypoints[k].pitch)));
               }
            else {
               waypoints[k].roll=acos(-1)+fabs(asin(waypoints[k].matrix[1][0]/cos(waypoints[k].pitch)));
               }
            }

     }

/* printout of waypoint matrices and orientation angles */
for(k=0;k<5;k++) {
    printf("\n");
    for(i=0;i<4;i++) {
            printf("\n");
            for(j=0;j<4;j++)
               printf("%8f ",waypoints[k].matrix[i][j]);
            }
            printf("%8f %8f %8f",waypoints[k].pitch,waypoints[k].yaw,waypoints[k].roll);
    }
cont=getch();


/*
            #define pi 3.1415926536
            typedef double homog4[4][4];

            homog4      *pos1,*pos2,*pos3,*posa,*pos_temp;
            double      deltab[6],deltac[6];
            double      T1,h,tacc,n;
            double a1,a2,a3,b1,b2,b3,a,c,

            pos1 = (homog4 *) malloc(sizeof(homog4));
            pos2 = (homog4 *) malloc(sizeof(homog4));
            pos3 = (homog4 *) malloc(sizeof(homog4));
            posa = (homog4 *) malloc(sizeof(homog4));

            for (i=0;i<4;i++)
               for (j=0;j<4;j++)
                     posa[i][j]=waypoints[0].matrix[i][j];

            for (i=0;i<4;i++)
               for (j=0;j<4;j++)
                     pos3[i][j]=waypoints[1].matrix[i][j];

            for (z=0;z<=z++) {
               pos_temp=pos1;
               pos1=posa;
               posa=pos_temp;

               pos_temp=pos2;
               pos2=pos3;
               pos3=pos_temp;

            for (i=0;i<4;i++)
               for (j=0;j<4;j++)
                     pos3[i][j]=waypoints[1].matrix[i][j];
*/
```

```
/* computation of smoothed trajectory points from waypoints */
/* initialize first point of trajectory to first waypoint */
i=1;
time=0;
tau=2*tacc;                          /* time is twice minimum acceleration time */
trajectories[0].xdist[0]=waypoints[0].xdist;
trajectories[0].ydist[0]=waypoints[0].ydist;
trajectories[0].zdist[0]=waypoints[0].zdist;
trajectories[0].pitch[0]=waypoints[0].pitch;
trajectories[0].yaw[0]=waypoints[0].yaw;
trajectories[0].roll[0]=waypoints[0].roll;
trajectories[0].time=0;

/* loop through three times using three waypoints to complete trajectory */
for (k=1;k<=3;k++) {
   deltab.xdist=waypoints[k].xdist-waypoints[k-1].xdist;
   deltab.ydist=waypoints[k].ydist-waypoints[k-1].ydist;
   deltab.zdist=waypoints[k].zdist-waypoints[k-1].zdist;
   deltab.pitch=waypoints[k].pitch-waypoints[k-1].pitch;
   deltab.roll=waypoints[k].roll-waypoints[k-1].roll;
   deltab.yaw=waypoints[k].yaw-waypoints[k-1].yaw;
   deltac.xdist=waypoints[k+1].xdist-waypoints[k].xdist;
   deltac.ydist=waypoints[k+1].ydist-waypoints[k].ydist;
   deltac.zdist=waypoints[k+1].zdist-waypoints[k].zdist;
   deltac.pitch=waypoints[k+1].pitch-waypoints[k].pitch;
   deltac.roll=waypoints[k+1].roll-waypoints[k].roll;
   deltac.yaw=waypoints[k+1].yaw-waypoints[k].yaw;
   a.xdist=waypoints[k-1].xdist;
   a.ydist=waypoints[k-1].ydist;
   a.zdist=waypoints[k-1].zdist;
   a.pitch=waypoints[k-1].pitch;
   a.roll=waypoints[k-1].roll;
   a.yaw=waypoints[k-1].yaw;
   distance=sqrt(pow(deltac.xdist,2)+pow(deltac.ydist,2)+pow(deltac.zdist,2));
   tvmax=distance/vmax;                                   /* time to travel leg at maximum velocity */
   tleg=(tvmax<2*tacc) ? 2*tacc : tvmax+2*tacc;      /* leg is either twice minimum acceleration time or tmaxvel */
   for(h=0;h<=1;j++) {
            h=h+.1;
            time=time+h*2*tau;
            trajectories[i].xdist[2]=(1/(2*pow(tau,2)))*(deltac.xdist*tau/tleg-deltab.xdist);
            trajectories[i].ydist[2]=(1/(2*pow(tau,2)))*(deltac.ydist*tau/tleg-deltab.ydist);
            trajectories[i].zdist[2]=(1/(2*pow(tau,2)))*(deltac.zdist*tau/tleg-deltab.zdist);
            trajectories[i].pitch[2]=(1/(2*pow(tau,2)))*(deltac.pitch*tau/tleg-deltab.pitch);
            trajectories[i].roll[2]=(1/(2*pow(tau,2)))*(deltac.roll*tau/tleg-deltab.roll);
            trajectories[i].yaw[2]=(1/(2*pow(tau,2)))*(deltac.yaw*tau/tleg-deltab.yaw);
            trajectories[i].xdist[1]=(h/tau)*(deltac.xdist*tau/tleg-deltab.xdist)+deltab.xdist/tau;
            trajectories[i].ydist[1]=(h/tau)*(deltac.ydist*tau/tleg-deltab.ydist)+deltab.ydist/tau;
            trajectories[i].zdist[1]=(h/tau)*(deltac.zdist*tau/tleg-deltab.zdist)+deltab.zdist/tau;
            trajectories[i].pitch[1]=(h/tau)*(deltac.pitch*tau/tleg-deltab.pitch)+deltab.pitch/tau;
            trajectories[i].roll[1]=(h/tau)*(deltac.roll*tau/tleg-deltab.roll)+deltab.roll/tau;
            trajectories[i].yaw[1]=(h/tau)*(deltac.yaw*tau/tleg-deltab.yaw)+deltab.yaw/tau;
            trajectories[i].xdist[0]=pow(h,2)*(deltac.xdist*tau/tleg-deltab.xdist)+deltab.xdist*2*h+a.xdist;
            trajectories[i].ydist[0]=pow(h,2)*(deltac.ydist*tau/tleg-deltab.ydist)+deltab.ydist*2*h+a.ydist;
            trajectories[i].zdist[0]=pow(h,2)*(deltac.zdist*tau/tleg-deltab.zdist)+deltab.zdist*2*h+a.zdist;
            trajectories[i].pitch[0]=pow(h,2)*(deltac.pitch*tau/tleg-deltab.pitch)+deltab.pitch*2*h+a.pitch;
            trajectories[i].roll[0]=pow(h,2)*(deltac.roll*tau/tleg-deltab.roll)+deltab.roll*2*h+a.roll;
            trajectories[i].yaw[0]=pow(h,2)*(deltac.yaw*tau/tleg-deltab.yaw)+deltab.yaw*2*h+a.yaw;
```

```
                    trajectories[i].time=time;
/*                  printf("\n\n%3d\t%8f",i,time);
                    for (j=2;j>=0;j--) {
                       printf("\n");
                       printf("%3d\n",j);
                       printf("xdist %8f\tydist %8f\tzdist %8f\npitch %8f\troll %8f\tyaw
%8f",trajectories[i].xdist[j],trajectories[i].ydist[j],trajectories[i].zdist[j],trajectories[i].pitch[j],trajectories[i].roll[j],trajectories[i].y
aw[j]);
                       }
                    contugetch(); */
                    }
            }

   display_trajectory();
   contugetch();

         close (fd);
         _setvideomode( _DEFAULTMODE );


}
/* ---------------------------------------------- */
void chain_code(struct object *object1,float alpha,int threshold,int buff,int fd)
   {
         XY_rgn_buf        rgn, rgn2, blank_line, blank_line2;
         int               status, flag, n;
         int               rstart, cstart, rowtemp, coltemp, nc;      /* holding variables for chain code start location */
         int               delta, length, dirout, dirin, decision;
         int               dirin1, dirout1;
         int               i, j, p, q, gamma;
         float             numerator, denominator, theta, perimeter, area;
         float m[4][4];
         float mu[4][4];
         float nu[4][4];
         float phi[5];
         char              s;
   static int dc[8][8] = {
         {0,0,0,0,3,3,3,3},
         {2,2,2,2,0,1,1,1},
         {2,2,2,2,0,0,1,1},
         {2,2,2,2,0,0,0,1},
         {2,2,2,2,0,0,0,0},
         {0,1,1,1,3,3,3,3},
         {0,0,1,1,3,3,3,3},
         {0,0,0,1,3,3,3,3}
   };
   length = 768;

   /* initialize arrays and variables */
   for(p=0;p<4;p++) {
         for(q=0;q<4;q++) {
            m[p][q]=0.0;
            mu[p][q]=0.0;
            nu[p][q]=0.0;
            }
         }
   for(p=0;p<=4;p++)
         phi[p]=0.0;
   perimeter=0.0;

   /* open specified buffer */
   if (buff==0)
         dt57_io_sel (fd, DT57_BUF0);   /* Prepare to talk to Buffer 0 */
   else
         dt57_io_sel (fd, DT57_BUF1);   /* Prepare to talk to Buffer 1 */
```

```
n = object1->col+100;
dirin = 1;

/* read first pixel */
rgn.region.x = object1->col;                    /* column 0 is left edge */
rgn.region.y = object1->row;                    /* y-coordinate is the ROW */
rgn.region.width = 1;                   /* entire line */
rgn.region.height = 1;                  /* only ONE row */
rgn.size = 1;                                   /* BUFFER 0 holds 1-byte elements */
rgn.buf = (char *) calloc (rgn.region.width*rgn.region.height,
                                                sizeof(char));
status = ioctl (fd, GET_XY_RGN, &rgn);          /* get the region */
if (status != E_NORMAL) {
        printf("Error reading line of image.  Exiting ...\n\n");
        exit(1);
        }


if (polarity == 1) {
   /* continue reading pixels until first light pixel is found */
   while (((0xff&((char *)rgn.buf)[0])<=threshold) && (object1->col<=n)) {
        object1->col++;
        rgn.region.x = object1->col;                    /* column 0 is left edge */
        rgn.region.y = object1->row;                    /* y-coordinate is the ROW */
        rgn.region.width = 1;                   /* entire line */
        rgn.region.height = 1;                  /* only ONE row */
        rgn.size = 1;                                   /* BUFFER 0 holds 1-byte elements */
        rgn.buf = (char *) calloc (rgn.region.width*rgn.region.height,
                                                sizeof(char));
        status = ioctl (fd, GET_XY_RGN, &rgn);  /* get the region */
        if (status != E_NORMAL) {
          printf("Error reading line of image.  Exiting ...\n\n");
          exit(1);
          }
        }
   }
else {
   while (((0xff&((char *)rgn.buf)[0])>=threshold) && (object1->col<=n)) {
        object1->col++;
        rgn.region.x = object1->col;                    /* column 0 is left edge */
        rgn.region.y = object1->row;                    /* y-coordinate is the ROW */
        rgn.region.width = 1;                   /* entire line */
        rgn.region.height = 1;                  /* only ONE row */
        rgn.size = 1;                                   /* BUFFER 0 holds 1-byte elements */
        rgn.buf = (char *) calloc (rgn.region.width*rgn.region.height,
                                                sizeof(char));
        status = ioctl (fd, GET_XY_RGN, &rgn);  /* get the region */
        if (status != E_NORMAL) {
          printf("Error reading line of image.  Exiting ...\n\n");
          exit(1);
          }
        }
   }



   if (object1->col>n)          /* no object found on this row */
        printf("\nNo object found on this row...");
   else {
        /* start chain code computation */
        nc=0;       /* number of elements in chain code */
        rstart=object1->row; /* holding variables for start of chain code */
        cstart=object1->col;
        printf("\n row %4d, col %4d",rstart, cstart);
        perimeter=0;
        flag=0;
```

```
do {
   /* Grab pixel */
   rgn.region.x = object1->col;                    /* column 0 is left edge */
   rgn.region.y = object1->row;                    /* y-coordinate is the ROW */
   rgn.region.width = 1;                  /* entire line */
   rgn.region.height = 1;                 /* only ONE row */
   rgn.size = 1;                                   /* BUFFER 0 holds 1-byte elements */
   rgn.buf = (char *) calloc (rgn.region.width*rgn.region.height,
                                                   sizeof(char));
   status = ioctl (fd, GET_XY_RGN, &rgn);          /* get the region */
   if (status != E_NORMAL) {
         printf("Error reading line of image.  Exiting ...\n\n");
         exit(1);
         }

   /* check to see if back at initial edge pixel */
   if (rstart==object1->row && cstart==object1->col && no>=2)
         flag=1;

   dirin = (dirin+2)%8;          /*initial search direction */
   rowtemp=object1->row;
   coltemp=object1->col;
   for (i=1;i<8;i++) {
         switch (dirin) {
               case 0:
                  object1->col++;
                  break;
               case 1:
                  object1->row--;
                  object1->col++;
                  break;
               case 2:
                  object1->row--;
                  break;
               case 3:
                  object1->row--;
                  object1->col--;
                  break;
               case 4:
                  object1->col--;
                  break;
               case 5:
                  object1->row++;
                  object1->col--;
                  break;
               case 6:
                  object1->row++;
                  break;
               case 7:
                  object1->row++;
                  object1->col++;
                  break;
               default:
                  break;
         }
         /* Grab pixel */
         rgn.region.x = object1->col;                    /* column 0 is left edge */
         rgn.region.y = object1->row;                    /* y-coordinate is the ROW */
         rgn.region.width = 1;                  /* entire line */
         rgn.region.height = 1;                 /* only ONE row */
         rgn.size = 1;                                   /* BUFFER 0 holds 1-byte elements */
         rgn.buf = (char *) calloc (rgn.region.width*rgn.region.height,
                                                   sizeof(char));
         status = ioctl (fd, GET_XY_RGN, &rgn);  /* get the region */
         if (status != E_NORMAL) {
            printf("Error reading line of image.  Exiting ...\n\n");
            exit(1);
```

```
            }

    if (polarity == 1) {
            if ((0xff&((char *)rgn.buf)[0])>threshold) {
               i=8;
               }
            else {
               dirin = (dirin+7)%8;
               object1->row=rowtemp;
               object1->col=coltemp;
               }
            }
    else {
            if ((0xff&((char *)rgn.buf)[0])<threshold) {
               i=8;
               }
            else {
               dirin = (dirin+7)%8;
               object1->row=rowtemp;
               object1->col=coltemp;
               }
            }

    }
            rgn2.region.x = rgn.region.x;                   /* column 0 is left edge */
            rgn2.region.y = rgn.region.y;                   /* y-coordinate is the ROW */
            rgn2.region.width = 1;                  /* entire line */
            rgn2.region.height = 1;                 /* only ONE row */
            rgn2.size = 1;                                  /* BUFFER 0 holds 1-byte elements */
            rgn2.buf = (char *) calloc (rgn.region.width*rgn.region.height,
                                                            sizeof(char));

            ((char *)rgn2.buf)[0] = 200;
                status = ioctl (fd, PUT_XY_RGN, &rgn2);             /* restore the region */
                if (status != E_NORMAL) {
                        printf("Error restoring line of image.  Exiting ...\n\n");
                        exit(1);
                        }
    nc++;               /* update #pixels in chain code */

if (nc>=2) {
   dirin1=dirout1;
   dirout1=dirin;
   decision=dc[dirin1][dirout1];
   switch (dirin1) {
            case 0:
               perimeter++;
               break;
            case 1:
               perimeter=perimeter+sqrt(1+pow(alpha,2));
               break;
            case 2:
               perimeter=perimeter+alpha;
               break;
            case 3:
               perimeter=perimeter+sqrt(1+pow(alpha,2));
               break;
            case 4:
               perimeter++;
               break;
            case 5:
               perimeter=perimeter+sqrt(1+pow(alpha,2));
               break;
            case 6:
               perimeter=perimeter+alpha;
               break;
            case 7:
               perimeter=perimeter+sqrt(1+pow(alpha,2));
```

```
                break;
            default:
                break;
            }
        switch (decision) {
            case 1:
                for(j=0;j<4;j++) {
                    m[i][0]=m[i][0]+pow((alpha*rowtemp),j);
                    m[i][1]=m[i][1]+pow((alpha*rowtemp),j)*coltemp;
                    m[i][2]=m[i][2]+pow((alpha*rowtemp),j)*pow(coltemp,2.0);
                    m[i][3]=m[i][3]+pow((alpha*rowtemp),j)*pow(coltemp,3.0);
                    }
                break;
            case 2:
                for(j=0;j<4;j++) {
                    m[i][0]=m[i][0]-(pow((alpha*rowtemp),j)*(coltemp-1.0));
                    m[i][1]=m[i][1]-(pow((alpha*rowtemp),j)*coltemp*(coltemp-1.0)/2.0);
                    m[i][2]=m[i][2]-(pow((alpha*rowtemp),j)*coltemp*(coltemp-1.0)*(2*coltemp-1.0)/6.0);
                    m[i][3]=m[i][3]-(pow((alpha*rowtemp),j)*pow((coltemp*(coltemp-1.0)/2.0),2.0));
                    }
                break;
            case 3:
                for(j=0;j<4;j++) {
                    m[i][0]=m[i][0]+pow((alpha*rowtemp),j)*coltemp;
                    m[i][1]=m[i][1]+pow((alpha*rowtemp),j)*coltemp*(coltemp+1.0)/2.0;
                    m[i][2]=m[i][2]+pow((alpha*rowtemp),j)*coltemp*(coltemp+1.0)*(2.0*coltemp+1.0)/6.0;
                    m[i][3]=m[i][3]+pow((alpha*rowtemp),j)*pow((coltemp*(coltemp+1.0)/2.0),2.0);
                    }
                break;
            default:
                break;
            }
    }
    else
        dirout1=dirin;

    } while (!flag);
    if (nc < 2)
            printf("\nNull chain code");

object1->xbar=m[1][0]/m[0][0];
object1->ybar=m[0][1]/m[0][0];
mu[0][0]=m[0][0];
mu[1][0]=0;
mu[0][1]=0;
mu[2][0]=m[2][0]-m[1][0]*object1->xbar;
mu[0][2]=m[0][2]-m[0][1]*object1->ybar;
mu[1][1]=m[1][1]-m[1][0]*object1->ybar;
mu[3][0]=m[3][0]-(3*object1->xbar*m[2][0])+(2*m[1][0]*(pow(object1->xbar,2)));
mu[1][2]=m[1][2]-(2*object1->ybar*m[1][1])-(object1->xbar*m[0][2])+(2*(pow(object1->ybar,2))*m[1][0]);
mu[2][1]=m[2][1]-(2*object1->xbar*m[1][1])-(object1->ybar*m[2][0])+(2*(pow(object1->xbar,2))*m[0][1]);
mu[0][3]=m[0][3]-(3*object1->ybar*m[0][2])+(2*pow(object1->ybar,2)*m[0][1]);

/* invariant moment computation for later object recognition */
for(p=0;p<4;p++) {
    for(q=0;q<4;q++) {
            gamma=(p+q)/2+1;
            nu[p][q]=mu[p][q]/pow(mu[0][0],gamma);
            }
    }
phi[1]=nu[2][0]+nu[0][2];
phi[2]=pow((nu[2][0]-nu[0][2]),2)+4*pow(nu[1][1],2);
phi[3]=pow((nu[3][0]-3*nu[1][2]),2)+pow((3*nu[2][1]-nu[0][3]),2);
phi[4]=pow((nu[3][0]+nu[1][2]),2)+pow((nu[2][1]+nu[0][3]),2);
/*  phi[5]=(nu[3][0]-3*nu[1][2])*(nu[3][0]+nu[1][2])*(pow((nu[3][0]-3*nu[1][2]),2)-3*pow((nu[2][1]+nu[0][3]),2))
      +(3*nu[2][1]-nu[0][3])*(nu[2][1]+nu[0][3])*(3*pow((nu[3][0]+nu[1][2]),2)-pow((nu[2][1]+nu[0][3]),2));
phi[6]=(nu[2][0]+nu[0][2])*(pow((nu[3][0]+nu[1][2]),2)-pow((nu[2][1]+nu[0][3]),2))
```

```
            +4*nu[1][1]*((nu[3][0]+nu[1][2])*(nu[2][1]+nu[0][3]));
        phi[7]=(3*nu[2][1]-nu[3][0])*(nu[3][0]+nu[1][2])*(pow((nu[3][0]+nu[1][2]),2)-3*pow((nu[2][1]+nu[0][3]),2))
            +(3*nu[1][2]-nu[3][0])*(nu[2][1]+nu[0][3])*(3*pow((nu[3][0]+nu[1][2]),2)-pow((nu[2][1]+nu[0][3]),2));
*/
        /* computation of object angle */
/*      numerator=2*nu[1][1];
        denominator=nu[2][0]-nu[0][2];
        if (numerator==0 && denominator< .0)
          theta=0;
        else
          theta=.5*atan2(denominator,numerator);
*/
        /* transfer of arrays into structure format */
        for(p=1;p<=4;p++)
          object1->phi[p]=phi[p];

        area=nu[0][0];
        object1->compactness=pow(perimeter,2)/area;
        printf("\n%6f is the perimeter of the object ",perimeter);
        printf("\n%6f is the compactness of the object ",object1->compactness);
        printf("\n(%6f,%6f) is the (row,col) of the centroid",object1->xbar,object1->ybar);
        }
  }
/* ---------------------------------------------------------- */
void get_point(int *colnum,int *rownum,int buff,int fd)
/*int colnum, rownum, buff, fd;*/                          /* mouse column and row, buffer number, device */
  {
        char            cont;                              /* continue flag, 0 to repeat */
        int             status, i, thresh;          /* */
        /* mouse routine variables */
        int             x, y;               /* old mouse position */
        int             x1, y1;     /* new mouse position */
        int             p1;                 /* button press/release indicator */
        int             p2, p3;     /* button press location */
        int             p4, p5;     /* button release location */
        int             mstatus;  /* mouse status indicator */
        int             flag;               /* flag for mouse button press */
        int             ok_flag=0;          /* flag for restoration of image */
        int             row, coll1, colr1;          /* row, left column, right column numbers */
        int             rowup2, rowdn2, col;                /* top row, bottom row, column number */
        XY_rgn_buf      rgn, rgn2, blank_line, blank_line2;

  if (buff==0)
        dt67_lo_sel (fd, DT67_BUF0);  /* Prepare to talk to Buffer 0 */
  else
        dt67_lo_sel (fd, DT67_BUF1);  /* Prepare to talk to Buffer 1 */

  /* Prepare to save and blank a line of the image */
  rgn.region.x = 0;                     /* column 0 is left edge */
  rgn.region.width = 768;               /* entire line */
  rgn.region.height = 1;                /* only ONE row */
  rgn.size = 1;                         /* BUFFER 0 holds 1-byte elements */
  rgn.buf = (char *) calloc (rgn.region.width*rgn.region.height,
                                                sizeof(char));
  blank_line.region.x = 0;              /* column 0 is left edge */
  blank_line.region.width = 768;        /* entire line */
  blank_line.region.height = 1;         /* only ONE row */
  blank_line.size = 1;                  /* BUFFER 0 holds 1-byte elements */
  blank_line.buf = (char *) calloc (blank_line.region.width*blank_line.region.height,
                                                sizeof(char));
  rgn2.region.x = 0;                    /* column 0 is left edge */
  rgn2.region.width = 1;                /* entire line */
  rgn2.region.height = 512;             /* only ONE row */
  rgn2.size = 1;                        /* BUFFER 0 holds 1-byte elements */
  rgn2.buf = (char *) calloc (rgn2.region.width*rgn2.region.height,
                                                sizeof(char));
  blank_line2.region.x = 0;             /* column 0 is left edge */
```

```
blank_line2.region.width = 1;                    /* entire line */
blank_line2.region.height = 512;                 /* only ONE row */
blank_line2.size = 1;                                    /* BUFFER 0 holds 1-byte elements */
blank_line2.buf = (char *) calloc (blank_line2.region.width*blank_line2.region.height,
                                                                        sizeof(char));
mouse_init();

row = 1;
x=-1; /* impossible values */
y=-1; /* impossible values */
flag=1;
hide_cursor();
do {
        do {
                x1=x;
                y1=y;
                get_mouse_xy( &mstatus, &x, &y );
                get_button_press(0, &mstatus, &p1, &p2, &p3);
                get_button_release(0, &mstatus, &p1, &p4, &p5);
                if (p1 != 0) {
                   *colnum=p4;
                   *rownum=p5;
                   flag=0;}
          } while (x==x1 && y==y1 && !p1);
        if (ok_flag==1) {
                 status = ioctl (fd, PUT_XY_RGN, &rgn);                 /* restore the region */
                 status = ioctl (fd, PUT_XY_RGN, &rgn2);                /* restore the region */
                 if (status != E_NORMAL) {
                        printf("Error restoring line of image.  Exiting ...\n\n");
                        exit(1);
                        }
                }
        ok_flag=1;

        if (x>=10 || x<=502) {
           colit1=x-10;
           colrt1=x+10;}
        else {
                if (x<10) {
                   colit1=x;
                   colrt1=x+10;}
                else {
                   colit1=x-10;
                   colrt1=x;}
        }
        row=y;

        if (y>=10 || y<=758) {
           rowup2=y-10;
           rowdn2=y+10;}
        else {
           if (y<10) {
                   rowup2=y;
                   rowdn2=y+10;}
           else {
                   rowup2=y-10;
                   rowdn2=y;}
        }
        col=x;

        rgn.region.y = row;                                     /* y-coordinate is the ROW */
        rgn.region.x = colit1;                          /* column 0 is left edge */
        rgn.region.width = colrt1-colit1+1;             /* entire line */
        blank_line.region.x = colit1;           /* column 0 is left edge */
        blank_line.region.width = colrt1-colit1+1;      /* entire line */
        status = ioctl (fd, GET_XY_RGN, &rgn);          /* get the region */
        if (status != E_NORMAL) {
```

```
            printf("Error reading line of image.  Exiting ...\n\n");
            exit(1);
            }

        rgn2.region.y = rowup2;                                 /* y-coordinate is the ROW */
        rgn2.region.x = col;                                    /* column 0 is left edge */
        rgn2.region.width = 1;                  /* entire line */
        rgn2.region.height = rowdn2-rowup2+1;
        blank_line2.region.x = col;             /* column 0 is left edge */
        blank_line2.region.height = rowdn2-rowup2+1;    /* entire line */
        status = ioctl (fd, GET_XY_RGN, &rgn2);         /* get the region */
        if (status != E_NORMAL) {
            printf("Error reading line of image.  Exiting ...\n\n");
            exit(1);
            }

        blank_line.region.y = rgn.region.y;
        for (i=0; i<=(colrt1-collt1+1); i++)
                ((char *)blank_line.buf)[i] = 255-((char *)rgn.buf)[i];
        status = ioctl (fd, PUT_XY_RGN, &blank_line);   /* blank the line */
        if (status != E_NORMAL) {
                printf("Error blanking line of image.  Exiting ...\n\n");
                exit(1);
                }


        blank_line2.region.y = rgn2.region.y;
        for (i=0; i<=(rowdn2-rowup2+1); i++)
                ((char *)blank_line2.buf)[i] = 255-((char *)rgn2.buf)[i];
        status = ioctl (fd, PUT_XY_RGN, &blank_line2);  /* blank the line */
                if (status != E_NORMAL) {
                    printf("Error blanking line of image.  Exiting ...\n\n");
                    exit(1);
                    }


    } while (!kbhit() && flag);
                        status = ioctl (fd, PUT_XY_RGN, &rgn);                  /* restore the region */
                        status = ioctl (fd, PUT_XY_RGN, &rgn2);                 /* restore the region */
                        if (status != E_NORMAL) {
                                printf("Error restoring line of image.  Exiting ...\n\n");
                                exit(1);
                                }

}
/* ---------------------------------------- */
void take_pix(int camera,int buff,int fd)
    /* Picture-taking procedure using Camera 0 or 1 */
    {
            int             i, thresh, status;
            char            cont;
            u_long          red[256], green[256], blue[256];

    if (camera==0)
            dt67_inp_chan (fd, DT67_CHAN0);             /* use camera 0 */
    else
            dt67_inp_chan (fd, DT67_CHAN1);             /* use camera 1 */

    if (buff==0) {
            dt67_io_sel (fd, DT67_BUF0);    /* Prepare to talk to Buffer 0 */
            dt67_inp_sel (fd, DT67_ACQ0);               /* picture goes to buffer 0 */
            dt67_disp_sel (fd, DT67_BUF0);              /* select buffer 0 for display */
            }
    else {
            dt67_io_sel (fd, DT67_BUF1);    /* Prepare to talk to Buffer 1 */
            dt67_inp_sel (fd, DT67_ACQ1);               /* picture goes to buffer 1 */
            dt67_disp_sel (fd, DT67_BUF1);              /* select buffer 1 for display */
```

```
        }

  dt67_inp_gain (fd, DT67_GAIN1);          /* set input a/d gain to 1 */
  dt67_disp_mask (fd, 0xFF);               /* set bit-wise display mask */
  dt67_passthru (fd);

  cont = 48;
  thresh=210;
  while (cont == 48) {
        printf("\n\nBinarization threshold (0-255): ");
        scanf(" %d",&thresh);

        dt67_stop (fd);
        for (i=0; i<256; i++)  /* Set ILUT values */
           if (i<thresh)
                green[i]=0;
           else
                green[i]=255;

        status = dt67_inp_lut (fd, DT67_WRITE, DT67_ILUT, DT67_LUT0, 0, 256, green);
        if (status != E_NORMAL)
           printf("Abnormal ILUT operation ...\n");
        dt67_passthru (fd);
        printf("\nis this threshold satisfactory? (0 to retry) ");
        contagetch();
        }
        dt67_stop (fd);
  }

/* ---------------------------------------------------------- */

void take_two_pix (fd)
int             fd;
{
        int     i;
        long    j;

  dt67_inp_gain (fd, DT67_GAIN1);          /* set input a/d gain to 1 */
  dt67_disp_mask (fd, 0xFF);               /* set bit-wise display mask */

        dt67_inp_chan (fd, DT67_CHAN0);          /* use camera 0 */
        dt67_inp_sel (fd, DT67_ACQ0);            /* picture goes to buffer 0 */
        dt67_disp_sel (fd, DT67_BUF0);           /* select buffer 0 for display */
        printf ("\nCAMERA 0 Live Image.  Press any key to freeze ...  ");
        dt67_passthru (fd);
        getch();
        dt67_stop (fd);                          /* take a picture */

        dt67_inp_chan (fd, DT67_CHAN1);          /* use camera 1 */
        dt67_inp_sel (fd, DT67_ACQ1);            /* picture goes to buffer 1 */
        dt67_disp_sel (fd, DT67_BUF1);           /* select buffer 1 for display */
        printf ("\nCAMERA 1 Live Image.  Press any key to freeze ...  ");
        dt67_passthru (fd);
        getch();
        dt67_stop (fd);                          /* take a picture */
        printf("\n\nToggling pictures.  Press any key to quit ...  ");
        for (i=0; !kbhit(); i=1-i) {
                if (i==0)
                        dt67_disp_sel (fd, DT67_BUF0);          /* select buffer 0 for display */
                else
                        dt67_disp_sel (fd, DT67_BUF1);          /* select buffer 1 for display */
                for (j=0; j<200000; j++)
                        ;
        }
        while (kbhit())
                getch();                /* use up keys in buffer */
}
```

```
/* ---------------------------------------- */
void init()
{
   aspect=4.0/3.0;
   scale=.11;
   dist=15000.0;
/*   if (!_setvideomode(_VRES16COLOR))
         puts("Can't enter _vres16color graphics mode"); */
   maxX=640;
   maxY=480;
   _setviewport(0,0,maxX,maxY);
   xoffset=maxX/2.0;
   xscale=(maxX/2.0)/scale/aspect;
   yoffset=maxY/1.5;
   yscale=(maxY/2.0)/scale;
   _setbkcolor(_LIGHTBLUE); /* LIGHTGREEN */
   }
/* ---------------------------------------- */
void compute_view_transform(float theta, float gamma, float matrix[3][3])
{
        float   st,ct,sg,cg;


   /* theta=elevation, gamma=azimuth */
   st = sin(theta);
   ct = cos(theta);
   sg = sin(gamma);
   cg = cos(gamma);

   /* transformation from 1 to 2 */
   matrix[0][0] = -st;
   matrix[1][0] = -ct*cg;
   matrix[2][0] = -ct*sg;
   matrix[0][1] = ct;
   matrix[1][1] = -st*cg;
   matrix[2][1] = -st*sg;
   matrix[0][2] = 0;
   matrix[1][2] = -sg;
   matrix[2][2] = cg;

   /* compute viewer's position in world coordinates */
   view_position[0]=cg*ct;
   view_position[1]=cg*st;
   view_position[2]=sg;

   }
/* ---------------------------------------- */
void compute_RPY_transform(float yaw, float pitch, float roll, float matrix[3][3])
{
        float   sp,cp,sr,cr,sy,cy;
        int     i;
        float   z_coord;


   sp = sin(pitch);
   cp = cos(pitch);
   sr = sin(roll);
   cr = cos(roll);
   sy = sin(yaw);
   cy = cos(yaw);

   matrix[0][0] = cy*cp;
   matrix[1][0] = sy*cp;
   matrix[2][0] = sp;
   matrix[0][1] = -sy*cr-cy*sp*sr;
   matrix[1][1] = cy*cr-sy*sp*sr;
   matrix[2][1] = cp*sr;
```

```
    matrix[0][2] = sy*sr-cy*sp*cr;
    matrix[1][2] = -cy*sr-sy*sp*cr;
    matrix[2][2] = cp*cr;

    }
/* ---------------------------------------- */
void compute_AzEl_transform(float az, float el, float matrix[3][3])
{
        float   sa,ca,se,ce;


    sa = sin(az);
    ca = cos(az);
    se = sin(el);
    ce = cos(el);

    matrix[0][0] = ce*ca;
    matrix[1][0] = -ce*sa;
    matrix[2][0] = se;
    matrix[0][1] = sa;
    matrix[1][1] = ca;
    matrix[2][1] = 0;
    matrix[0][2] = -se*ca;
    matrix[1][2] = se*sa;
    matrix[2][2] = ce;

    }
/* ---------------------------------------- */
void transform_point(int tr, float p1[3], int p0[2])
{
        int     i,j,k;                  /* loop counters */
        float   perspective_scale;
        float   p_temp[3],p2[3];
        float   p_intermediate[2];

    for (k=tr;k>=0;k--) {               /* perform all transfroms from tr to 0 */
        for (i=0;i<=2;i++) {
            p_temp[i]=0;
            for (j=0;j<=2;j++)
                p_temp[i]=p_temp[i]+AA[k].matrix[i][j]*p1[j];
            }
        }

    /* next perform perspective transformation into 2-D */
    perspective_scale=1/(p_temp[1]+dist);
    p_intermediate[0]=p_temp[0]*perspective_scale;  /* perspective view x-coord */
    p_intermediate[1]=p_temp[2]*perspective_scale;  /* perspective view y-coord */

    /* now scale 2-D data to fit viewport */
    /* make sure p0 is an INTEGER */
    p0[0]=p_intermediate[0]*xscale+xoffset;
    p0[1]=-p_intermediate[1]*yscale+yoffset;
}
/* ---------------------------------------- */
void draw_all_figures()
{
        int             i,j,z;  /* loop counters */

/*  _setlinestyle(0xFFFF); */

/* Plot of shuttle trajectory */
    for (i=0;i<=26;i++) {
            path[i].matrix[0]=trajectories[i].ydist[0];
            path[i].matrix[1]=trajectories[i].zdist[0];
            path[i].matrix[2]=trajectories[i].xdist[0];
            transform_point(1,path[i].matrix,pathplot[i].matrix);
            printf("\n%6f,%6f",pathplot[i].matrix[0],pathplot[i].matrix[1]);
```

```
        }
        cont=getch();
        _clearscreen(_GCLEARSCREEN);
        color = 14;
        _setcolor(color);      /* GRAY */
        _moveto(pathplot[0].matrix[0],pathplot[0].matrix[1]);
        for(i=1;i<=25;i++) {
           _lineto(pathplot[i].matrix[0],pathplot[i].matrix[1]);
           cont=getch();
           }



   for (i=0;i<=3;i++) {
           for (j=0;j<=2;j++) {
              pts[i].matrix[j]=0;
              }
           }

   pts[1].matrix[0]=500;
   pts[2].matrix[1]=500;
   pts[3].matrix[2]=500;
   axis_text[1]="x";
   axis_text[2]="y";
   axis_text[3]="z";

   transform_point(0,pts[0].matrix,p2a.matrix);        /* origin of both coordinate frames */

   /* world coordinate axes */
   color = 12;
   _setcolor(color);  /* LIGHTCYAN */
   for(i=1;i<=3;i++) {
           _moveto(p2a.matrix[0],p2a.matrix[1]);
           transform_point(0,pts[i].matrix,p2b.matrix);
           _lineto(p2b.matrix[0],p2b.matrix[1]);
           _outgtext(axis_text[i]);
           }
   }
/* ---------------------------------------- */
void display_trajectory()
{
        int         i;

   init();
   view_theta_home=45.0;
   view_gamma_home=30.0;
   view_theta=view_theta_home;
   view_gamma=view_gamma_home;
   for(i=0;i<=4;i++)
           angles[i]=0;

/*   _clearscreen(_GCLEARSCREEN); */
   compute_view_transform(view_theta, view_gamma, AA[0].matrix);
   compute_RPY_transform(angles[2],angles[0],angles[1],AA[1].matrix);
   compute_AzEl_transform(angles[3],angles[4],AA[2].matrix);
   draw_all_figures();
   }

/* ---------------------------------------- */
/* ---------------------------------------- */
void classify ()
{
   int i;
        float n_out[8], net;

          for (i=0;i<5;i++)
```

```
                printf("\n%6f",data_in[i]);
  /* scale input values */
  n_out[0] = (  .3223281e+01) * data_in[0] + ( -.4076804e+00);
  n_out[1] = (  .5715583e+01) * data_in[1] + ( -.2884305e-02);
  n_out[2] = (  .1429102e+04) * data_in[2] + ( 0.0000000e+00);
  n_out[3] = (  .1733984e+04) * data_in[3] + ( 0.0000000e+00);
  n_out[4] = (  .2307651e-01) * data_in[4] + ( -.2991185e+00);

  net = ( -.8023978e+02);
  net += ( -.2527917e+02) * n_out[0];
  net += ( -.1068764e+02) * n_out[1];
  net += ( -.1874744e+02) * n_out[2];
  net += ( -.2874733e+02) * n_out[3];
  net += ( -.3174827e+02) * n_out[4];
  printf("net = %6f",net);
  n_out[5] = 1 / (1+exp(-net));

  net = ( -.3726717e+02);
  net += ( -.2117750e+02) * n_out[0];
  net += ( -.1000000e+03) * n_out[1];
  net += (  .3441722e+02) * n_out[2];
  net += (  .3320605e+02) * n_out[3];
  net += (  .1000000e+03) * n_out[4];
  net += ( -.1911363e+02) * n_out[5];
  printf("net = %6f",net);
  n_out[6] = 1 / (1+exp(-net));

  net = ( -.2853116e+02);
  net += (  .1000000e+03) * n_out[0];
  net += (  .1000000e+03) * n_out[1];
  net += (  .1000000e+03) * n_out[2];
  net += (  .1000000e+03) * n_out[3];
  net += ( -.8579510e+02) * n_out[4];
  net += ( -.3038764e+02) * n_out[5];
  printf("net = %6f",net);
  n_out[7] = 1 / (1+exp(-net));

  /* scale output values */
  data_out[0] = ( n_out[6] - ( 0.0000000e+00) ) / (  .9999999e+00);
  data_out[1] = ( n_out[7] - ( 0.0000000e+00) ) / (  .9999999e+00);
  printf("\nOutput 1 = %6f, Output 2 = %6f",data_out[0],data_out[1]);
  printf("\nObject is Numeral %6f",(2.0*data_out[0]+data_out[1]));

}
/* ---------------------------------------- */
```

```
/* "mouse_i.c" -- module of mouse routines based on calls to "int 51".
            W. Clement, 8/91. */
```

```
/* ---------------------------------------- */
/* ---------------------------------------- */
mouse_init()
{
        flag_reset ( &status, &num_buttons );
        if ( status )
                printf("Mouse driver NOT INSTALLED.\n\n");
        else {
                driver_installed_flag = 1;
                set_mouse_xy ( 10000, 10000 );          /* choose very large numbers */
                get_mouse_xy ( &status, &MAX_x, &MAX_y );
                flag_reset ( &status, &num_buttons );   /* reset to center */
                show_cursor();
                set_horiz_limits( 0, 773 );
                set_vert_limits( 0, 517 );
        }
}
/* ---------------------------------------- */
choose_mouse_cursor(status,x,y)         /* selects mouse shape to use based on */
int             *status, *x, *y;                /* location and returns coordinates */
{
        if (!driver_installed_flag)
                return(0);
        get_mouse_xy ( status, x, y );

}
/* ---------------------------------------- */




/*      "flag_reset()" is used to determine whether the mouse driver has been
        installed and, if so, the number of buttons on the mouse.  Also, the
        mouse's position is reset to the middle of the screen after this call.
        Upon return:
                If mouse driver NOT installed:
                        status=0, num_buttons=0
                If mouse driver IS installed:
                        status=-1, num_buttons=number of mouse buttons
*/
/* ---------------------------------------- */
flag_reset( status, num_buttons)
int     *status, *num_buttons;
{
        regs.x.ax = 0;                  /* Mouse Driver system call 0 */
        int86(MOUSE_INT, &regs, &regs);
        *status = regs.x.ax;
        *num_buttons = regs.x.bx;
}
/* ---------------------------------------- */


/* Show or Hide the mouse cursor. */
/* ---------------------------------------- */
show_cursor()
{
        if (!driver_installed_flag)
                return(0);
        regs.x.ax = 1;                  /* Mouse Driver system call 1 */
        int86(MOUSE_INT, &regs, &regs);
}
/* ---------------------------------------- */
hide_cursor()
{
```

```
        if (!driver_installed_flag)
                return(0);
        regs.x.ax = 2;                  /* Mouse Driver system call 2 */
        int86(MOUSE_INT, &regs, &regs);
}
/* ---------------------------------------- */


/*      "get_mouse_xy()" returns the horizontal (x) and vertical (y) pixel
        number.  "set_mouse_xy()" repositions the mouse cursor.
*/
/* ---------------------------------------- */
get_mouse_xy ( status, x, y )
int     *status, *x, *y;
{
        if (!driver_installed_flag)
                return(0);
        regs.x.ax = 3;                  /* Mouse Driver system call 3 */
        int86(MOUSE_INT, &regs, &regs);
        *status = regs.x.bx;
        *x = regs.x.cx;
        *y = regs.x.dx;
}
/* ---------------------------------------- */
set_mouse_xy ( x, y )
int     x, y;
{
        if (!driver_installed_flag)
                return(0);
        regs.x.ax = 4;                  /* Mouse Driver system call 4 */
        regs.x.cx = x;
        regs.x.dx = y;
        int86(MOUSE_INT, &regs, &regs);
}
/* ---------------------------------------- */


/* "get_button_press()" and "get_button_release()" return the same values
        for "status" regardless of the value sent in variable "button":
                status=0  -> no button pressed.
                status=1  -> left button pressed.
                status=2  -> right button pressed.
                status=3  -> both buttons pressed.
        However, the "x" and "y" positions are tied to the "button" you choose.
                button=0  -> left mouse button
                button=1  -> right mouse button
        For "get_button_press()", the values returned in "x" and "y" are the
        horizontal and vertical pixel values of the last position at which the
        button was pressed.
                For "get_button_release()", the values returned in "x" and "y" are the
        horizontal and vertical pixel values of the last position at which the
        button was released.
                "press_count" and "release_count" are, respectively, the number of
        times the designated button was pressed or released since the last call
        to this function.
*/
/* ---------------------------------------- */
get_button_press ( button, status, press_count, x, y )
int     button, *status, *press_count, *x, *y;
{
        if (!driver_installed_flag)
                return(0);
        regs.x.ax = 5;                  /* Mouse Driver system call 5 */
        regs.x.bx = button;
        int86(MOUSE_INT, &regs, &regs);
```

```
        *status = regs.x.ax;
        *press_count = regs.x.bx;
        *x = regs.x.cx;
        *y = regs.x.dx;
}
/* ------------------------------------------------ */
get_button_release ( button, status, release_count, x, y )
int     button, *status, *release_count, *x, *y;
{
        if (!driver_installed_flag)
                return(0);
        regs.x.ax = 6;                  /* Mouse Driver system call 6 */
        regs.x.bx = button;
        int86(MOUSE_INT, &regs, &regs);
        *status = regs.x.ax;
        *release_count = regs.x.bx;
        *x = regs.x.cx;
        *y = regs.x.dx;
}
/* ------------------------------------------------ */


/* "set_horiz_limits" sets the inclusive range of values the cursor can
        take in the horizontal (x) direction.  This value specifies the
        location of the upper left pixel in the 16x16 pixel mouse cursor
        (in graphics mode).
        "set_vert_limits" performs similarly.
*/
/* ------------------------------------------------ */
set_horiz_limits ( minx, maxx )
int     minx, maxx;
{
        if (!driver_installed_flag)
                return(0);
        regs.x.ax = 7;                  /* Mouse Driver system call 7 */
        regs.x.cx = minx;
        regs.x.dx = maxx;
        int86(MOUSE_INT, &regs, &regs);
}
/* ------------------------------------------------ */
set_vert_limits ( miny, maxy )
int     miny, maxy;
{
        if (!driver_installed_flag)
                return(0);
        regs.x.ax = 8;                  /* Mouse Driver system call 8 */
        regs.x.cx = miny;
        regs.x.dx = maxy;
        int86(MOUSE_INT, &regs, &regs);
}
/* ------------------------------------------------ */

screen_on()
{       _setvideomode( _VRES16COLOR );
}

screen_off()
{       _setvideomode( _DEFAULTMODE );
}
```